Package 'rankimp'

September 17, 2026

Title Consensus Ranking of Variable Importance with Uncertainty

Version 1.0.0

Description Variable importance rankings depend on the method, the random seed and the resample used to compute them. This package treats every source of importance as a judge expressing a ranking over the predictors, and synthesises those rankings into a Kemeny median ranking with ties. Uncertainty about the consensus is quantified through bootstrap rank confidence sets, top-k probabilities and clustering of disagreeing judges.

License MIT + file LICENSE

URL https://github.com/agostinognasso/rankimp

BugReports https://github.com/agostinognasso/rankimp/issues

Depends R (>= 3.5)

Imports ConsRank, ggplot2, stats, tibble, utils

Suggests cluster, covr, kernelshap, knitr, randomForest, ranger, rmarkdown, rsample, stabm, testthat (>= 3.1.7)

VignetteBuilder knitr

LazyData true

Config/testthat/edition 3

Encoding UTF-8

Language en-GB

Config/roxygen2/version 8.1.0

NeedsCompilation no

Author Agostino Gnasso [aut, cre, cph] (ORCID: <https://orcid.org/0000-0002-8046-3923>)

Maintainer Agostino Gnasso <agostino.gnasso@unina.it>

Repository CRAN

Date/Publication 2026-09-17 11:30:02 UTC

Contents

`applications` *Synthetic loan applications, with the right answer attached*

Description

Eight hundred loan applications, seven predictors and a default indicator. Generated rather than collected, so that every question about a ranking of the predictors has an answer to be checked against rather than argued over.

Usage

```
applications
```

Format

A data frame with 800 rows and 8 columns:

income Annual income at application.

bureau_score Credit bureau score, 300 to 850.

debt_ratio Debt-to-income ratio at origination.

employment_yrs Years in the current job.

prior_arrears Arrears in the previous two years. A count with six distinct values.

credit_lines Open credit lines. Enters the outcome nowhere.

age Age in years. Enters the outcome nowhere.

default Did the loan default? A factor, `"no"` or `"yes"`, 23.0 per cent `"yes"`.

The right answer

Every predictor enters a linear predictor on the standardised scale, so a coefficient and an effect are the same number and the ordering is unambiguous. It is stored on the data frame:

```
attr(applications, "effects")
#>  prior_arrears     debt_ratio   bureau_score         income
#>           0.90           0.78           0.62           0.45
#> employment_yrs   credit_lines            age
#>           0.22           0.00           0.00
```

The attribute is dropped by subsetting, as attributes on a data frame are. Take a copy of it before you slice.

Why the methods disagree on it

Two things were built in to make them, because a dataset on which every importance measure agrees has nothing to say about a package for reconciling them.

`bureau_score` and `income` are built from one latent creditworthiness and correlate at 0.84, so they stand in for each other and the credit for the signal has to be divided somehow. Both measures below invert them relative to the truth.

`prior_arrears` is the largest effect in the data and takes six distinct values. Impurity importance is biased against predictors with few split points, and on a forest of 500 trees it puts `prior_arrears` **fourth** while permutation importance puts it **first**, which is where it belongs. Over the seven predictors the two measures agree at a Kendall tau of 0.71 with each other, and against the truth at 0.59 for impurity and 0.88 for permutation.

That is the disagreement this package exists to handle: not noise, not a tie, but two respectable measures ordering the same variables differently because they are answering slightly different questions.

Both measures do agree on the two predictors that do nothing, which end up last. A procedure that ranked either of them above a signal variable would be reporting noise, and `rank_confsets()` should decline to order them.

Source

Generated by `inst/data-raw/applications.R`, which is shipped with the package and records the specification as code. Seeded and re-runnable. Realistic in shape and invented in fact.

See Also

`importance_judges()` to build a panel on it, `rank_confsets()` to put uncertainty around the consensus.

Examples

```
truth <- attr(applications, "effects")
sort(truth, decreasing = TRUE)

set.seed(1)
fit <- randomForest::randomForest(default ~ ., data = applications,
```

```
                                  ntree = 200, importance = TRUE)

# Impurity puts the largest effect fourth; permutation puts it first.
rank(-importance_mdi(fit))[["prior_arrears"]]
rank(-importance_permutation(fit, applications, "default"))[["prior_arrears"]]
```

`autoplot.consensus_rank` *Visualise a consensus ranking against the panel it came from*

Description

The consensus rank of each variable, drawn on top of every rank the judges actually gave it. Point area is the number of judges at that rank, so the picture is exact rather than jittered.

Usage

```
## S3 method for class 'consensus_rank'
autoplot(object, ...)
```

Arguments

`object`	A `consensus_rank` object.
`...`	Reserved for future use.

Details

What it is for: a consensus ranking reports one number per variable, and that number is equally consistent with a panel that agreed and a panel that was split down the middle. The spread behind each point is the difference, and it is per variable. `tau_x` and `item_consensus()` measure agreement per *judge*, which is a different question and will not tell you *which* variables the panel could not place.

Variables the consensus could not separate come out at the same rank and are drawn at the same height; that is a finding, not a drawing artefact. Where the Kemeny median is not unique the plot plots the combined ranking, the one `consensus_rank()` reports, and says so in the subtitle.

The rank axis is reversed, so rank 1, the most important variable, sits at the top.

Value

A `ggplot` object.

See Also

`consensus_rank()`, `item_consensus()`, `autoplot.rank_confsets()`

Examples

```
judges <- rbind(
  permutation = c(1, 2, 3, 4), impurity = c(1, 3, 2, 4), loco = c(2, 1, 3, 4)
)
colnames(judges) <- c("income", "age", "balance", "region")
autoplot(consensus_rank(judges))
```

`autoplot.judge_clusters` *Visualise the disagreement between judges*

Description

Multidimensional scaling of the judges in the space of the Kemeny-Snell distance, coloured by cluster.

Usage

```
## S3 method for class 'judge_clusters'
autoplot(object, ...)
```

Arguments

`object`	A `judge_clusters` object.
`...`	Reserved for future use.

Details

What the picture is for: whether the panel is one cloud or several, and which judges sit between them. The Kemeny-Snell distance is integer-valued and rarely Euclidean, so two dimensions are a projection and not the thing itself. The subtitle reports how much of the distance survives the projection, and a low figure means the plot is a sketch of the grouping rather than evidence for it.

Value

A `ggplot` object.

See Also

`judge_clusters()`

Examples

```
judges <- rbind(
  permutation_1 = c(1, 2, 3, 4, 5, 6), permutation_2 = c(1, 2, 3, 4, 6, 5),
  permutation_3 = c(2, 1, 3, 4, 5, 6), impurity_1    = c(6, 5, 4, 3, 2, 1),
  impurity_2    = c(5, 6, 4, 3, 2, 1), impurity_3    = c(6, 5, 4, 3, 1, 2)
)
colnames(judges) <- c("income", "age", "balance", "region", "tenure", "arrears")
autoplot(judge_clusters(judges))
```

---

autoplot.rank_confsets

Visualise the rank confidence sets

Description

Variables ordered by consensus rank, each with the interval of ranks it plausibly occupies. Overlapping intervals are the honest way of saying that two variables cannot be ordered on this evidence, which is the statement most importance plots decline to make.

Usage

```
## S3 method for class 'rank_confsets'
autoplot(object, ...)
```

Arguments

`object`	A `rank_confsets` object.
`...`	Reserved for future use.

Details

The rank axis is reversed, so that rank 1, the most important variable, sits at the top.

Value

A `ggplot` object.

See Also

`rank_confsets()`

Examples

```
judges <- rbind(c(1, 2, 3, 4), c(1, 3, 2, 4), c(2, 1, 3, 4))
colnames(judges) <- c("income", "age", "balance", "region")
autoplot(rank_confsets(consensus_rank(judges), n_boot = 50))
```

consensus_rank *Kemeny consensus ranking of variable importance*

Description

Given K judges, each expressing a ranking over the same p variables, returns the median ranking in the sense of Kemeny: the ranking that minimises the total Kemeny-Snell distance to the judges,

$$\pi^* = \arg\min_{\pi} \sum_{k=1}^{K} w_k \, d_{KS}(\pi, \pi_k).$$

Usage

```
consensus_rank(
  x,
  weights = NULL,
  algorithm = c("auto", "exact", "quick", "fast", "decor"),
  ties = TRUE
)

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

Arguments

x	A consensus_rank object.
weights	Optional numeric vector of length nrow(x) giving the weight of each judge. Weights let a permutation importance computed out-of-bag count for more than an impurity-based one. When x is a judges object carrying method weights (see importance_judges() and judge_weights()), those are used unless weights overrides them.
algorithm	One of "auto", "exact", "quick", "fast", "decor".
ties	Keep ties in the consensus ranking.
...	Unused.

Details

Ties in the consensus are meaningful and are kept by default. Variables that the judges genuinely cannot separate *should* come out equal, which is what distinguishes a Kemeny median from an average of Borda scores. Set ties = FALSE to force a linear order.

Value

An object of class consensus_rank, a list with elements ranking (a tibble of variable and consensus rank), tau (the average tau_x agreement between the consensus and the judges), consensus_all

(every optimal consensus found, one per row), `judges` and `weights` (the panel as supplied), and the settings used.

When several rankings attain the minimum, `ranking` holds their combination rather than an arbitrary one of them: see the section below.

The panel is kept because the consensus alone is a point estimate: `rank_confsets()` resamples the judges to put an interval around it, and it cannot do that from a ranking.

When the median is not unique

Several rankings can attain the same minimum, and `ConsRank` returns them all. Reporting one of them would be arbitrary in a way that is not neutral: which comes first depends on the order of the columns, so a variable can gain a position by sitting to the left. That was measured: on a symmetric panel, permuting the columns changed the winner; in a simulation with three exchangeable noise predictors the leftmost took the best rank systematically, and the situation is not rare, arising in 57% to 98% of replicates there.

The consensus reported is therefore the optimal set combined: each variable takes its average position over the optima, and those that come out equal are tied. Variables the objective genuinely cannot separate are reported as equal, which is the point of taking a median over weak orderings. The whole set remains in `consensus_all`.

Choice of algorithm

Finding the Kemeny median is NP-hard, so `algorithm = "auto"` picks by problem size:

- `p <= 10`: `"exact"`, branch-and-bound.
- `11 <= p <= 50`: `"quick"`.
- `p > 50`: `"fast"`, with a message. The integer-programming route of the roadmap, which would restore optimality guarantees at this size, is a phase F2 deliverable.

The exact threshold is ten rather than the fifteen `ConsRank` permits, because the cost of branch-and-bound is not a smooth function of `p` and the panels this package produces are the hard ones. Importance scores tie: the unimportant variables all sit near zero and rank equal. On tied panels of thirty judges the same solver took 0.010 s at `p = 10`, 0.78 s at `p = 11` and 280 s at `p = 12`. Meanwhile `"quick"` returned the identical consensus and the identical `tau_x` on twenty out of twenty tied panels at `p = 10`. Exactness above ten variables buys little and can cost minutes, so ask for it deliberately with `algorithm = "exact"`.

See Also

`importance_to_rank()`, `rank_confsets()`

Examples

```
judges <- rbind(
  c(1, 2, 3, 4),
  c(1, 2, 4, 3),
  c(2, 1, 3, 4)
)
colnames(judges) <- c("income", "age", "balance", "region")
```

```
consensus_rank(judges)
```

`importance_backends` *Importance backends*

Description

Each backend asks a fitted model for one named numeric vector of importance scores over its predictors, higher meaning more important. They are the judges' voices: `importance_judges()` calls them once per judge, and they can equally be called directly when a single score vector is all that is needed.

Usage

```
importance_permutation(fit, data, target, n_perm = 5L, ...)

importance_mdi(fit, data = NULL, target = NULL, ...)

importance_shap(fit, data, target, n_explain = 100L, bg_n = 200L, ...)

importance_loco(fit, data, target, newdata = NULL, ...)
```

Arguments

`fit`	A fitted model (randomForest or ranger).
`data`	A data frame holding the response and every predictor the model was trained on. For `importance_mdi()` the scores are read off the fit, so `data` and `target` are accepted but unused.
`target`	Name of the response column in `data`.
`n_perm`	Number of independent shuffles per predictor.
`...`	Ignored. It absorbs the arguments meant for the other backends when the call comes from `importance_judges()`, which forwards its own `...` to every backend it invokes.
`n_explain`	Maximum number of rows to explain.
`bg_n`	Size of the background sample, passed to `kernelshap::kernelshap()`.
`newdata`	Optional data frame on which the losses are evaluated.

Details

Permutation, MDI and LOCO are implemented natively against the supported engines (randomForest, ranger); SHAP delegates to the kernelshap package. Scores are computed on the data supplied, so permutation and LOCO are in-sample unless `data` is a holdout set. `importance_judges()` with a `resamples` axis is the out-of-sample version.

Backends that randomise (permutation shuffles, SHAP row subsampling, LOCO refits) draw from the session RNG: call `set.seed()` first for reproducibility.

Value

A named numeric vector of importance scores, one per predictor.

Functions

- `importance_permutation()`: Permutation importance: the mean increase in loss (RMSE for regression, misclassification rate for classification) when one predictor's column is shuffled, over `n_perm` shuffles.
- `importance_mdi()`: Mean decrease in impurity, read off the fit: `IncNodePurity` (regression) or `MeanDecreaseGini` (classification) for randomForest, `variable.importance` for a ranger model fitted with `importance = "impurity"`.
- `importance_shap()`: Mean absolute SHAP value, via `kernelshap::kernelshap()`. For classifiers the mean is also taken across classes; ranger classifiers must be fitted with `probability = TRUE`. At most `n_explain` rows of `data` are explained (sampled without replacement when there are more), against a background sample of `bg_n` rows drawn by kernelshap.
- `importance_loco()`: Leave-one-covariate-out: the increase in loss when the model is refitted on `data` without one predictor (preserving the original number of trees and `mtry`, capped) and compared with `fit`. Losses are evaluated on `newdata` when supplied, on `data` otherwise; `data` should be the data `fit` was trained on, or the comparison mixes training sets. The most expensive backend: one refit per predictor.

See Also

`importance_judges()`, `importance_to_rank()`

Examples

```
set.seed(1)
fit <- randomForest::randomForest(mpg ~ ., data = mtcars, ntree = 50)
importance_permutation(fit, mtcars, "mpg", n_perm = 2)


set.seed(1)
fit <- randomForest::randomForest(mpg ~ ., data = mtcars, ntree = 50)
importance_mdi(fit)


set.seed(1)
small <- mtcars[c("mpg", "wt", "hp", "qsec")]
fit <- randomForest::randomForest(mpg ~ ., data = small, ntree = 30)
importance_loco(fit, small, "mpg")
```

`importance_judges` *Assemble the panel of judges*

Description

Builds the ranking matrix consumed by `consensus_rank()` from any combination of the four axes along which a variable importance ranking can vary: the *method* used, the *model* it interrogates, the *seed* of the ensemble, and the *resample* of the data. Every combination of the active axes becomes one judge, one row of the panel, whose importance scores are computed by the matching backend and turned into a ranking with `importance_to_rank()`.

Usage

```
importance_judges(
  fit_list,
  methods = c("permutation", "mdi"),
  data = NULL,
  target = NULL,
  resamples = NULL,
  seeds = NULL,
  weights = NULL,
  ties_method = c("min", "average", "first"),
  ...
)

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

Arguments

`fit_list`	A fitted model, or a (preferably named) list of them. Supported engines: randomForest, ranger. Unnamed models are called `model1`, `model2`, ...
`methods`	Character vector of importance methods, among `"permutation"`, `"mdi"`, `"shap"`, `"loco"`.
`data`	Data frame holding the response and the predictors. Required unless `methods` is only `"mdi"` and no `seeds` or `resamples` are given.
`target`	Name of the response column in `data`.
`resamples`	Optional `rset` (from `rsample::vfold_cv()` or `rsample::bootstraps()`) giving the resampling axis.
`seeds`	Optional integer vector of seeds for the refitting axis.
`weights`	Optional named numeric vector of method weights, e.g. `c(permutation = 1, mdi = 0.5)`. Every method in `methods` must be named. The expanded per-judge weights travel with the panel and are picked up by `consensus_rank()`.
`ties_method`	Passed to `importance_to_rank()`.
`...`	Unused.
`x`	A `judges` object.

Value

An object of class `judges`: an integer ranking matrix with one row per judge, carrying as attributes the `provenance` of each row (a tibble with the judge's model, engine, method, seed and resample), the raw `scores` behind the ranks, the per-judge `weights` (or `NULL`), and the `recipe` that built it.

The four axes

- **Methods** (`methods`): each importance method is one voice; see importance_backends.
- **Models** (`fit_list`): several fitted models, so that a cross-model consensus asks which variables matter regardless of the learner. All models must be trained on the same predictors.
- **Seeds** (`seeds`): each seed refits every model on `data` after `set.seed(seed)`, measuring the stability of the ranking under the ensemble's own randomness. Refits keep the original number of trees and `mtry`.
- **Resamples** (`resamples`): an `rset` from the rsample package (v-fold CV or bootstrap). Every split refits the models on its analysis set; permutation and SHAP importance are then computed on the assessment set, and LOCO refits on the analysis set and evaluates on the assessment set, giving the out-of-sample versions of those importances. MDI, which has no data argument, is read off the refit.

Without `seeds` or `resamples` the supplied fits are used as they are, and data-dependent importances are in-sample. Axes combine as a full grid: models x methods x seeds x resamples.

The recipe

The panel keeps the arguments that built it: the fits, the data, the target, the methods, the axes and the backend settings. That is what lets `rank_confsets()` rebuild it on a bootstrap sample of the rows without being handed them all again. That makes the panel as large as the objects it refers to.

Reproducibility

Permutation shuffles, SHAP row subsampling and refits draw from the session RNG; call `set.seed()` before this function for a reproducible panel.

The `seeds` axis sets seeds of its own, and puts the stream back where it found it, so building a panel does not move the caller's RNG. That is more than politeness: `rank_confsets()` with `type = "data"` rebuilds the panel once per bootstrap replicate and draws the next resample from this same stream, and a panel that parked it made every replicate resample the same rows.

See Also

importance_backends, `judge_weights()`, `consensus_rank()`

Examples

```
set.seed(1)
fit <- randomForest::randomForest(mpg ~ ., data = mtcars, ntree = 50)
judges <- importance_judges(fit, methods = c("permutation", "mdi"),
                            data = mtcars, target = "mpg", n_perm = 2)
judges
consensus_rank(judges)
```

importance_to_rank *Turn importance scores into rankings*

Description

Each row of `x` holds the importance that one judge assigns to each variable; the result holds the rank that judge gives each variable, with 1 for the most important. Variables a judge scores equally receive the same rank, because pretending to separate them would invent information the judge never supplied.

Usage

```
importance_to_rank(x, ties_method = c("min", "average", "first"))
```

Arguments

`x`	Numeric matrix or data frame of importance scores, judges in rows and variables in columns. Higher is more important.
`ties_method`	Passed to `base::rank()`. The default, `"min"`, produces the weak orderings that the Kemeny framework expects.

Value

An integer matrix of rankings with the same dimensions and dimnames as `x`, suitable for `consensus_rank()`.

Examples

```
imp <- rbind(
  permutation = c(income = 0.31, age = 0.12, balance = 0.12),
  impurity    = c(income = 0.44, age = 0.20, balance = 0.05)
)
importance_to_rank(imp)
```

item_consensus *Agreement of each judge with the consensus*

Description

The Emond-Mason `tau_x` between each judge's ranking and the consensus. `cr$tau` is the (weighted) mean of this column; the column itself says whether that mean summarises a panel that agrees or averages a panel that is split, which are different situations reported by the same number.

Usage

```
item_consensus(cr)
```

Arguments

cr	A consensus_rank object.

Details

A judge with a tau_x near zero is not necessarily wrong. Marginal and conditional importance measures disagree by construction when the predictors are correlated, and one of them will look like an outlier against a panel dominated by the other.

Value

A tibble with one row per judge: its name (or index), its weight, and its tau_x against the consensus, in increasing order of agreement.

See Also

judge_clusters() for what to do when the agreement is low.

Examples

```
judges <- rbind(
  permutation = c(1, 2, 3, 4),
  shap        = c(1, 3, 2, 4),
  impurity    = c(4, 3, 2, 1)
)
colnames(judges) <- c("income", "age", "balance", "region")
item_consensus(consensus_rank(judges))
```

---

judge_clusters *Do the judges agree?*

Description

When the global agreement with the consensus is low, reporting the consensus alone hides the disagreement instead of describing it. Clustering the judges in the space of the Kemeny-Snell distance recovers the sub-populations of methods that see the model differently. Marginal against conditional importance measures typically separate here whenever the predictors are correlated, and that separation is itself the finding.

Usage

```
judge_clusters(
  judges,
  k = NULL,
  weights = NULL,
  algorithm = c("auto", "exact", "quick", "fast", "decor"),
  n_null = 199L,
  alpha = 0.05,
```

```
  seed = 1L,
  medoid_limit = 20000L
)

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

Arguments

judges	A judges object or a ranking matrix.
k	Number of clusters, or NULL to select it automatically.
weights	Optional per-judge weights, one per row. Taken from a judges panel when it carries them. They weigh each group's consensus, not the distances.
algorithm	Solver used for each group's consensus, passed to consensus_rank().
n_null	Panels drawn from a single population to judge the observed grouping against. 0 skips the test, and the largest silhouette then wins outright. Ignored when k is given.
alpha	How rarely a single population must group as sharply as this panel does before the panel is called divided.
seed	Seed for the reference panels. The caller's random stream is restored afterwards.
medoid_limit	Largest number of candidate medoid sets to enumerate before falling back on a greedy start.
x	A judge_clusters object.
...	Unused.

Details

The groups are k-medians in ranking space: each group's centre is the Kemeny median of its members, computed by consensus_rank(), and each judge belongs to the group whose centre it is closest to. The centre of a group is therefore a consensus ranking, the object worth reporting, rather than a point in some embedding.

Value

An object of class judge_clusters, a list with elements clustering (a tibble of judge, cluster and silhouette width), cluster (the same assignment as a named integer vector), k, centres (the group consensus rankings, one per row), consensus (the consensus_rank object behind each centre), distances (the Kemeny-Snell distances between judges), criterion (the silhouette at every k the search could have used, reported so the shape of the panel can be inspected; the automatic choice is the test's, not this column's maximum), test (the observed statistic, its p-value against one population, and the spread the reference panels were given), and the settings used.

Why this returns the same answer twice

Nothing here draws from the RNG. The starting partition is the exactly optimal set of medoids, found by enumerating every one of them while the panel is small enough to allow it, and the refinement is deterministic; ties are broken on the lowest index. A panel of judges is small, so the

usual reason for random restarts, an initialisation too expensive to optimise, does not apply, and an inference function that answered differently on every call would be worth less than the answer it gives.

Above `medoid_limit` candidate sets the enumeration is replaced by a greedy choice, which is still deterministic but no longer certified optimal; the returned object says which was used.

Whether to split at all

The panel is left whole unless it groups more sharply than a single population of judges would. That test is not decoration. The average silhouette width on its own divides a homogeneous panel far too readily, because two judges who happen to rank alike sit at distance zero and score a silhouette of exactly 1: on eight variables and six judges one transposition apart, the silhouette alone split a single population 62% of the time. Those same zero distances are what makes a real division obvious, so the statistic cannot tell the two cases apart by itself. It has to be told what one population looks like.

So the panel's best split in two is compared against the best split in two of `n_null` panels drawn from one population, spread to match the mean distance between the judges actually supplied. The panel is divided only when fewer than `alpha` of those reference panels split as sharply. What the test costs in divisions missed, and what it buys in divisions not invented, is measured in `inst/simulations/cluster-recovery.R`.

What it is measured to do

From that script, 300 panels per cell of eight variables. A panel drawn from **one population** is divided anyway 2.0% of the time at six judges (4.0% when those judges are noisier), 5.3% at ten and 8.3% at sixteen, against a nominal `alpha` of 5%. Two well-separated populations are recovered exactly 0.877 of the time at six judges, 0.873 at ten and 0.943 at sixteen; on groups that are close, or judges that are noisy, it falls a long way: 0.360 at six judges with the group centres four transpositions apart, and 0.073 when the judges stray three.

On real panels of permutation against LOCO judges over correlated predictors, 100 datasets per cell, the panel divides in two 23 times in 100 at a correlation of 0.9 with six judges and 82 times in 100 with sixteen. All 23 of the six-judge divisions fell exactly on the method families; 69 of the 82 at sixteen judges did. Panel size is what buys sensitivity here, and six judges, which is two methods by three seeds, has little of it. `vignette("method-disagreement")` works through one of the panels that does not divide.

The hypothesis is "one population", so the statistic is the best division in two, not the best over every `k`. Testing the maximum over `k` sounds more general and is worse: the reference pays a multiplicity that grows with the panel, and power falls away as judges are added. Measured on two clearly separated groups at the same level: 0.233 at six judges down to 0.067 at twelve for the maximum, against 0.633 up to 0.917 for the two-group statistic.

A panel the test rejects is therefore reported as divided **in two**, which is the division the evidence is about. Reading `k` off the largest silhouette instead attaches an uncalibrated number to a calibrated decision, and it measures worse: on two separated groups the partition is recovered exactly 0.943 of the time at sixteen judges against 0.690 for the largest silhouette, at the same false division rate. It is also what made larger panels perform worse: recovery fell from 0.877 at six judges to 0.690 at sixteen, and now rises to 0.943. Pass `k` explicitly to fit any other number; a panel that genuinely holds three groups is reported as two.

All of it is read off the distances alone, through the exactly optimal medoid partitions, which is what makes hundreds of reference panels affordable. Which judge goes where, and what each

group ranks, is the k-medians refinement of that partition.

Reproducibility

The reference panels are drawn from `seed`, and the session's random stream is put back where it was found: a call neither depends on the stream nor disturbs it, and two calls on the same panel agree down to the p-value.

See Also

`item_consensus()` for the same question asked one judge at a time, `autoplot.judge_clusters()` to see the panel in Kemeny-Snell space.

Examples

```
# Three judges who rank by one logic, three by another. Four variables would
# not be enough for the test to call it: with 24 possible rankings a panel
# groups this sharply by chance often enough to matter.
judges <- rbind(
  permutation_1 = c(1, 2, 3, 4, 5, 6), permutation_2 = c(1, 2, 3, 4, 6, 5),
  permutation_3 = c(2, 1, 3, 4, 5, 6), impurity_1    = c(6, 5, 4, 3, 2, 1),
  impurity_2    = c(5, 6, 4, 3, 2, 1), impurity_3    = c(6, 5, 4, 3, 1, 2)
)
colnames(judges) <- c("income", "age", "balance", "region", "tenure", "arrears")

het <- judge_clusters(judges)
het
split(rownames(judges), het$cluster)
```

`judge_weights` *Weights for the panel of judges*

Description

Not every judge deserves an equal say. Impurity-based importance is known to favour high-cardinality predictors, so a panel that mixes it with permutation importance may want to down-weight it rather than let the two cancel out. The returned vector plugs straight into `consensus_rank()`'s `weights` argument.

Usage

```
judge_weights(judges, by = c("equal", "method", "reliability"), values = NULL)
```

Arguments

judges	A judges object from importance_judges(), or a plain ranking matrix (judges in rows) for the schemes that need no provenance.
by	Weighting scheme: "equal", "method" or "reliability".
values	Named numeric vector of method weights, when by = "method": every method present in the panel must be named.

Value

A named numeric vector of positive weights, one per judge.

Schemes

- "equal": every judge weighs 1.
- "method": one weight per importance method, supplied through values and expanded over the judges. Needs the provenance that importance_judges() records, so it only works on a judges object.
- "reliability": a judge's weight grows with its agreement with the rest of the panel: $w_k = (1 + \bar{\tau}_k)/2$, where $\bar{\tau}_k$ is the mean Emond-Mason τ_x correlation between judge k and every other judge, computed with ConsRank::tau_x(). Weights are normalised to mean 1, and floored at machine epsilon so that a judge in perfect disagreement with everyone is effectively, though not numerically, excluded. Down-weighting the dissenters sharpens the consensus around the majority view; when dissent is the interesting signal, look at judge_clusters() instead of weighting it away.

See Also

importance_judges(), consensus_rank()

Examples

```
judges <- rbind(
  c(1, 2, 3, 4), c(1, 2, 4, 3), c(2, 1, 3, 4), c(4, 3, 2, 1)
)
colnames(judges) <- c("income", "age", "balance", "region")
judge_weights(judges, by = "reliability")
```

---

prob_topk *Probability that a variable lands in the top k*

Description

The proportion of bootstrap replicates in which the variable's consensus rank is at most k. Ties are counted as membership: a variable tied at rank k with another is in the top k.

Usage

```
prob_topk(cb, k = 5L)
```

Arguments

cb	A rank_confsets object.
k	Size of the top set.

Value

A tibble of variables and probabilities, in decreasing order of probability.

How much to believe the number

It is conservative in the middle and honest at the ends. Measured on 300 panels of eight predictors with close effects and eighty rows, pooled over k = 1..6 and binned by the reported probability (inst/simulations/select-calibration.R):

reported	actually in the top k
0.15	0.18
0.35	0.41
0.44	0.56
0.55	0.64
0.75	0.83
0.98	0.98

A variable given 0.44 is in the top k about 56% of the time, so the number understates by up to twelve points where it is least decisive, and is accurate where it is near 0 or near 1. The direction is the one to want, since the function never claims more than it can show, and the cause is the one behind the wide intervals: a replicate sees about 0.632n distinct rows, ranks the variables worse than the full sample does, and drops some of them out of the top k more often than the sampling distribution would.

See Also

rank_confsets()

Examples

```
judges <- rbind(c(1, 2, 3, 4), c(1, 3, 2, 4), c(2, 1, 3, 4))
colnames(judges) <- c("income", "age", "balance", "region")
prob_topk(rank_confsets(consensus_rank(judges), n_boot = 50), k = 2)
```

rank_confsets *Bootstrap confidence sets for the consensus ranking*

Description

A single consensus ranking is a point estimate. Resampling gives the sampling distribution of each variable's position, from which follow the interval of plausible ranks for each variable and the probability that a variable belongs to the top k.

Usage

```
rank_confsets(
  cr,
  n_boot = NULL,
  level = 0.95,
  type = c("judges", "data"),
  algorithm = c("quick", "exact", "fast", "decor")
)

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

Arguments

cr	A consensus_rank object.
n_boot	Number of bootstrap replicates. Defaults to 500 for type = "judges" and 50 for type = "data", which refits every model on every replicate.
level	Coverage of the rank confidence sets.
type	What to resample: the "judges" in the panel, or the "data" the models were fitted to.
algorithm	Solver used for the replicates. "quick" by default; "exact", "fast" and "decor" are also accepted, as in consensus_rank().
x	A rank_confsets object.
...	Unused.

Details

This is the inferential layer. It is what distinguishes the package from a rank-averaging exercise, and it is what lets a claim about variable importance be falsified: "income is the third most important variable" cannot be checked, while "income is in the top five with probability 0.97" can.

Value

An object of class rank_confsets, a list with elements confsets (a tibble of variable, consensus rank, and the lower and upper ends of the rank interval), ranks (the n_boot x p matrix of bootstrap ranks), level, n_boot, type, n_units (how many judges or rows were resampled), failed (replicates dropped) and consensus (the cr it came from).

What is resampled

Two questions, two answers, and they are not the same question.

`type = "judges"` draws the judges with replacement from `cr$judges`. It measures how much the consensus depends on *which sources of importance happened to be in the panel*, which a panel of ten permutation replicates and one SHAP judge will show. Resampling is done by drawing multinomial counts and passing them as judge weights rather than by materialising the resampled panel. The two are equivalent, since `ConsRank` treats a weight of 3 exactly as three copies of the judge, and the weighted form avoids rebuilding a `K x p` matrix per replicate. Judge weights supplied to `consensus_rank()` are carried through by multiplying them into the bootstrap counts.

`type = "data"` draws the *rows* with replacement, refits every model on the resampled data and rebuilds the whole panel from scratch, once per replicate. It measures how much the consensus depends on the sample the models were fitted to, which is the question a reader asks when they wonder whether the ranking would survive another dataset. It needs a panel built by `importance_judges()`, whose recipe carries the fits, the data and the settings; a plain ranking matrix has no models to refit.

How a data replicate is built

Each replicate draws `n` rows with replacement, refits every model on them with `importance_judges()`'s own machinery, and measures importance the way the recipe did: on the rows the bootstrap left out when the original panel judged out of sample (the `resamples` axis), on the resampled rows themselves when it judged in sample. Mirroring the recipe is what keeps the bootstrap distribution centred on the point estimate, without which a percentile interval means nothing.

Two consequences worth stating plainly:

- A `resamples` axis is **replaced** by the bootstrap's own in-bag/out-of-bag split, so a panel of `models x methods x V` judges is rebuilt with `models x methods` of them. Each replicate votes with fewer judges than the point estimate did, which makes its consensus noisier and the intervals, if anything, wider.
- Refits preserve the number of trees and `mtry` and nothing else. A forest fitted with a hand-tuned `nodesize` is refitted at the engine's default, here as on the `seeds` and `resamples` axes.

One consequence to expect rather than to debug: a replicate's ranking drifts towards the middle. `n` rows drawn with replacement hold about $0.632n$ distinct ones, and a variable is harder to place on that much less information, so the top of the ranking drifts down and the bottom drifts up. Measured over 300 panels of eight predictors with close effects and eighty rows (`inst/simulations/select-calibration.R`), the median bootstrap rank of the second variable of the consensus sits 0.55 ranks below it and is worse than it in 51% of panels; the eighth sits 1.00 above. Hand a replicate all the distinct rows instead and it returns the point estimate exactly, which is how this was told apart from a panel rebuilt wrongly.

The drift is not large enough to push the consensus rank out of its own interval: that happened in none of those 2,400 variable-replicates. Earlier versions of this page warned that it would, on the strength of one panel whose interval was `[3, 5]` around a consensus rank of 2. That panel came from the bootstrap that repeated its resamples, and the intervals it produced were too narrow to be believed.

A replicate that fails is dropped rather than allowed to kill the run, and the count of dropped replicates is warned about and kept in `failed`. The failure that actually happens is a response class too rare to survive a bootstrap draw: `randomForest` refuses to refit on a sample that lost one (`ranger`

drops the level and carries on). A rare *predictor* level is harmless, because subsetting a factor keeps its levels.

What the level actually buys

The two bootstraps miss the nominal level in opposite directions, and by enough that the choice between them is the choice that matters. The simulation is `inst/simulations/rank-coverage.R`: eight predictors, five of them real, 300 replicates per cell, nominal 0.95, coverage of the true rank of the signal variables.

- `type = "data"`: 0.966, 0.969 and 0.978 at `n` of 80, 200 and 500 with effects close enough that the middle of the ranking is barely identifiable; 0.993 and 0.998 at `n` of 80 and 200 with effects that separate cleanly.
- `type = "judges"`: 0.582, 0.655 and 0.711 on those same close cells; 0.801 and 0.929 on the separated ones.

The data bootstrap covers, and it covers by being wide. On the hardest cell its interval spans 5.1 of the 8 available ranks. It reports that this sample does not order these variables, which is the truth: the point estimate recovers the true order of the five signal variables in 5% of replicates there. An interval that admitted less would be claiming more than the data holds.

The judge bootstrap is narrow on the same cell, 2.0 ranks, and misses the true rank two times in five. Resampling a panel measures how much the methods disagree with each other, and that is not how far the ranking would move on another sample. It is the cheap answer to a different question, and the gap it leaves is the reason `type = "data"` exists: 0.966 against 0.582 where the ordering is hardest.

Fifty replicates are enough for the data bootstrap. The same cell at `n_boot = 200` covers 0.970 against 0.966, a difference inside the Monte Carlo error of either. That is worth stating because it was not always true of this package: a defect that made the bootstrap repeat its resamples once made `n_boot` look decisive (see `NEWS.md`).

Read a rank confidence set as a statement about what the evidence rules out, not as a calibrated guarantee. The rank is a discrete, non-smooth functional and a percentile bootstrap is not automatically valid for such functionals; the figures above are measured on those designs, not promised in general.

On the cost

Every replicate solves a Kemeny problem, which is NP-hard. Branch-and-bound is fast on panels that agree and pathological on panels that do not: with twelve variables and many ties, the normal case for importance scores where unimportant variables all tie near zero, a single exact solve has been measured at over four minutes, which is a day and a half for five hundred replicates.

The default is therefore to resample with the `"quick"` heuristic regardless of the algorithm used for the point estimate, and to say so. Pass `algorithm = "exact"` if the panel is small and you want optimality guarantees inside the bootstrap too.

A data replicate additionally refits every model and recomputes every importance, which is orders of magnitude dearer than reweighting a panel that already exists. Hence the smaller default for `n_boot`, and a message reporting the projected cost when the run looks like a long one.

See Also

prob_topk(), rank_select(), autoplot.rank_confsets()

Examples

```
judges <- rbind(
  c(1, 2, 3, 4), c(1, 2, 3, 4), c(1, 3, 2, 4),
  c(2, 1, 3, 4), c(1, 2, 4, 3), c(2, 1, 4, 3)
)
colnames(judges) <- c("income", "age", "balance", "region")
cb <- rank_confsets(consensus_rank(judges), n_boot = 50)
cb


# Resampling the data instead: every replicate refits the forest.
set.seed(1)
fit <- randomForest::randomForest(mpg ~ ., data = mtcars, ntree = 50)
J <- importance_judges(fit, methods = c("permutation", "mdi"),
                       data = mtcars, target = "mpg", n_perm = 2)
rank_confsets(consensus_rank(J), n_boot = 10, type = "data")
```

---

rank_select *Select variables with a rank guarantee*

Description

Keeps the variables whose whole rank confidence set lies at or above `threshold`, that is, whose upper (worst) end is no worse than `threshold`.

Usage

```
rank_select(cb, threshold = 10L)
```

Arguments

cb	A `rank_confsets` object.
threshold	Worst rank a selected variable may plausibly occupy.

Details

Selecting on the point estimate of the rank ignores that the ranking was estimated in the first place. Selecting on the confidence set makes the claim "this variable really is among the most important" one that the data can refuse: a variable whose interval straddles the threshold is not selected, and the reason is visible.

The rule is deliberately conservative. It answers "which variables am I sure about", not "which variables should I keep": a variable excluded here may still carry signal, and prob_topk() quantifies how much doubt there is.

Value

A character vector of selected variable names, in consensus order.

What the guarantee is measured to be worth

Conservative is a claim, so it was measured. On 300 panels of eight predictors with close effects and eighty rows (`inst/simulations/select-calibration.R`), against the true ordering of the data-generating coefficients:

threshold	selected per panel	false selections	panels with one	of those that deserved it, selected
1	1.00	0.000	0.000	1.000
2	1.01	0.000	0.000	0.503
3	1.07	0.009	0.010	0.352
4	1.23	0.011	0.013	0.305
5	1.56	0.002	0.003	0.312
6	2.22	0.030	0.067	0.431

A selected variable is almost never one that did not deserve it: at most 3% of selections, and 0% at the thresholds that make the strongest claim. The price is on the other side: past a threshold of 1 it selects between a third and a half of the variables that did deserve it. Read a short list as "these I can defend", never as "these are the ones that matter".

See Also

`rank_confsets()`, `prob_topk()`

Examples

```
judges <- rbind(c(1, 2, 3, 4), c(1, 2, 3, 4), c(1, 2, 4, 3))
colnames(judges) <- c("income", "age", "balance", "region")
cb <- rank_confsets(consensus_rank(judges), n_boot = 50)

rank_select(cb, threshold = 2)
prob_topk(cb, k = 2)
```

Index