Package ‘comperank’

October 12, 2022

Title Ranking Methods for Competition Results

Version 0.1.1

Description Compute ranking and rating based on competition results. Methods of different nature are implemented: with fixed Head-to-Head structure, with variable Head-to-Head structure and with iterative nature. All algorithms are taken from the book 'Who’s #1?: The science of rating and ranking' by Amy N. Langville and Carl D. Meyer (2012, ISBN:978-0-691-15422-0).

License MIT + file LICENSE

URL `https://github.com/echasnovski/comperank`

BugReports `https://github.com/echasnovski/comperank/issues`

Depends comperes (>= 0.1.0), R (>= 3.4.0)

Imports dplyr (>= 0.6.0), Rcpp, rlang (>= 0.2.0), tibble

Suggests covr, knitr, rmarkdown, testthat

LinkingTo Rcpp

VignetteBuilder knitr

Encoding UTF-8

LazyData true

RoxygenNote 7.0.2

NeedsCompilation yes

Author Evgeni Chasnovski [aut, cre]

Maintainer Evgeni Chasnovski <`evgeni.chasnovski@gmail.com`>

Repository CRAN

Date/Publication 2020-03-03 23:10:02 UTC

R topics documented:

`comperank-package` *comperank: Ranking and Rating Methods for Competition Results*

Description

`comperank` provides tools for computing ranking and rating based on competition results. It is tightly connected to its data infrastructure package comperes. Basic knowledge about creating valid competition results and Head-to-Head expressions with `comperes` is needed in order to efficiently use `comperank`.

Details

`comperank` leverages the tidyverse ecosystem of R packages. Among other things, it means that the main output format is tibble.

`comperank` gets inspiration from the book "Who's #1" by Langville and Meyer. It provides functionality for the following rating algorithms:

- Algorithms with **fixed Head-to-Head structure**:
  - Simplified Massey method with `rate_massey()` and `rank_massey()`.
  - Simplified Colley method with `rate_colley()` and `rank_colley()`.
- Algorithms with **variable Head-to-Head structure**:
  - Keener method with `rate_keener()` and `rank_keener()`.
  - Markov method with `rate_markov()` and `rank_markov()`.
  - Offense-Defense method with `rate_od()` and `rank_od()`.
- Algorithms with **iterative nature**:
  - General Iterative ratings with `rate_iterative()`, `rank_iterative()`, and `add_iterative_ratings()`.
  - Elo ratings with `rate_elo()`, `rank_elo()`, and `add_elo_ratings()`.

`comperank` also offers data sets describing professional snooker in seasons 2016/2017 and 2017/2018. See snooker_events, snooker_players, snooker_matches.

To learn more about `comperank` browse vignettes with browseVignettes(package = "comperank").

Author(s)

Maintainer: Evgeni Chasnovski `<evgeni.chasnovski@gmail.com>`

See Also

Useful links:

- `https://github.com/echasnovski/comperank`
- Report bugs at `https://github.com/echasnovski/comperank/issues`

`colley` *Colley method*

Description

Functions to compute rating and ranking using Colley method.

Usage

```
rate_colley(cr_data)

rank_colley(cr_data, keep_rating = FALSE, ties = c("average", "first",
  "last", "random", "max", "min"), round_digits = 7)
```

Arguments

`cr_data`	Competition results in format ready for as_longcr().
`keep_rating`	Whether to keep rating column in ranking output.
`ties`	Value for `ties` in `round_rank()`.
`round_digits`	Value for `round_digits` in `round_rank()`.

Details

This rating method was initially designed for games between two players. There will be an error if in `cr_data` there is a game not between two players. Convert input competition results manually or with to_pairgames() from `comperes` package.

It is assumed that score is numeric and higher values are better for the player.

Computation is done based only on the games between players of interest (see Players). **Note** that it isn't necessary for all players of interest to be present in `cr_data` but it might be a good idea in order to obtain plausible outcome rating.

The outline of the Colley method is as follows:

1. Compute Colley matrix: diagonal elements are equal to number of games played by certain player *plus* 2, off-diagonal are equal to minus number of common games played. This matrix will be the matrix of system of linear equations (SLE).
2. Compute right-hand side of SLE as 1 + 0.5*("number of player's wins" - "number of player's losses").
3. Solve the SLE. The solution is the Colley rating.

Value

`rate_colley()` returns a tibble with columns `player` (player identifier) and `rating_colley` (Colley rating). The mean rating should be 0.5. **Bigger value indicates better player performance**.

`rank_colley()` returns a `tibble` with columns `player`, `rating_colley` (if `keep_rating = TRUE`) and `ranking_colley` (Colley ranking computed with `round_rank()`).

Players

`comperank` offers a possibility to handle certain set of players. It is done by having `player` column (in longcr format) as factor with levels specifying all players of interest. In case of factor the result is returned only for players from its levels. Otherwise - for all present players.

References

Wesley N. Colley (2002) *Colley's Bias Free College Football Ranking Method: The Colley Matrix Explained*. Available online at `http://www.colleyrankings.com`

Examples

```
rate_colley(ncaa2005)

rank_colley(ncaa2005)

rank_colley(ncaa2005, keep_rating = TRUE)
```

`elo` *Elo method*

Description

Functions to compute rating and ranking using Elo method.

Usage

```
rate_elo(cr_data, K = 30, ksi = 400, initial_ratings = 0)

rank_elo(cr_data, K = 30, ksi = 400, initial_ratings = 0,
  keep_rating = FALSE, ties = c("average", "first", "last", "random", "max",
  "min"), round_digits = 7)

add_elo_ratings(cr_data, K = 30, ksi = 400, initial_ratings = 0)

elo(rating1, score1, rating2, score2, K = 30, ksi = 400)
```

Arguments

`cr_data`	Competition results in format ready for as_longcr().
`K`	K-factor for Elo formula.
`ksi`	Normalization coefficient for Elo formula.
`initial_ratings`	Initial ratings (see Iterative ratings).
`keep_rating`	Whether to keep rating column in ranking output.
`ties`	Value for `ties` in `round_rank()`.
`round_digits`	Value for `round_digits` in `round_rank()`.
`rating1`	Rating of player1 before the game.
`score1`	Score of player1 in the game.
`rating2`	Rating of player2 before the game.
`score2`	Score of player2 in the game.

Details

`rate_elo()` and `add_elo_ratings()` are wrappers for `rate_iterative()` and `add_iterative_ratings()` correspondingly. Rate function is based on Elo algorithm of updating ratings:

1. Probability of player1 (with rating r1) winning against player2 (with rating r2) is computed based on rating difference and sigmoid function: `P = 1 / (1 + 10^( (r2 - r1) / ksi ) )`. `ksi` defines the spread of ratings.
2. Result of the game from player1 perspective is computed based on rule: `S = 1` (if `score1 > score2`), `S = 0.5` (if `score1 == score2`) and `S = 0` (if `score1 < score2`).
3. Rating delta is computed: `d = K * (S - P)`. The more the `K` the more the delta (with other being equal).
4. New ratings are computed: `r1_new = r1 + d`, `r2_new = r2 - d`.

`elo()` function implements this algorithm. It is vectorized over all its arguments with standard R recycling functionality. **Note** that not this function is used in `rate_elo()` and `add_elo_ratings()` because of its not appropriate output format, but rather its non-vectorized reimplementation is.

Ratings are computed based only on games between players of interest (see Players) and `NA` values.

Value

`rate_elo()` returns a tibble with columns `player` (player identifier) and `rating_elo` (Elo ratings, based on row order, by the end of competition results). **Bigger value indicates better player performance**.

`rank_elo()` returns a `tibble` with columns `player`, `rating_elo` (if `keep_rating = TRUE`) and `ranking_elo` (Elo ranking computed with `round_rank()`).

`add_elo_ratings()` returns a widecr form of `cr_data` with four rating columns added:

- **rating1Before** - Rating of player1 before the game.
- **rating2Before** - Rating of player2 before the game.

- **rating1After** - Rating of player1 after the game.
- **rating2After** - Rating of player2 after the game.

`elo()` always returns a matrix with two columns containing ratings after the game. Rows represent games, columns - players.

Players

`comperank` offers a possibility to handle certain set of players. It is done by having `player` column (in longcr format) as factor with levels specifying all players of interest. In case of factor the result is returned only for players from its levels. Otherwise - for all present players.

References

Wikipedia page for Elo rating system.

Examples

```
# Elo ratings
rate_elo(ncaa2005)

rank_elo(ncaa2005)

rank_elo(ncaa2005, keep_rating = TRUE)

add_elo_ratings(ncaa2005, initial_ratings = 100)

# Elo function
elo((0:12)*100, 1, 0, 0)
elo((0:12)*100, 1, 0, 0, K = 10)
elo((0:12)*10, 1, 0, 0, ksi = 40)
```

`iterative` *Iterative rating method*

Description

Functions to compute Iterative numeric ratings, i.e. which are recomputed after every game, and corresponding rankings.

Usage

```
rate_iterative(cr_data, rate_fun, initial_ratings = 0)

rank_iterative(cr_data, rate_fun, initial_ratings = 0, keep_rating = FALSE,
  type = "desc", ties = c("average", "first", "last", "random", "max",
  "min"), round_digits = 7)

add_iterative_ratings(cr_data, rate_fun, initial_ratings = 0)
```

Arguments

`cr_data`	Competition results in format ready for as_longcr().
`rate_fun`	Rating function (see Details).
`initial_ratings`	Initial ratings (see Details).
`keep_rating`	Whether to keep rating column in ranking output.
`type`	Value for `type` in `round_rank()`: `"desc"` or `"asc"`.
`ties`	Value for `ties` in `round_rank()`.
`round_digits`	Value for `round_digits` in `round_rank()`.

Details

Iterative ratings of group of players are recomputed after every game based on players' game scores and their ratings just before the game. Theoretically this kind of ratings can be non-numeric and be computed on competition results with variable number of players but they rarely do. This package provides functions for computing iterative **numeric** ratings for pairgames (competition results with games only between two players). Error is thrown if `cr_data` is not pairgames.

Games in widecr form are arranged in increasing order of values in column `game` (if it is present) and processed from first to last row.

`NA` values in column `player` are allowed. These players are treated as 'ghosts': players of the same rating as opponent before the game. 'Ghosts' are not actual players so they don't appear in the output of `rate_iterative()`. For games between two 'ghosts' ratings before and after the game are set to 0.

The core of the rating system is `rate_fun`. It should take the following arguments:

- **rating1** - Rating of player1 before the game.
- **score1** - Score of player1 in the game.
- **rating2** - Rating of player2 before the game.
- **score2** - Score of player2 in the game.

`rate_fun` should return a numeric vector of length 2: first element being a rating of player1 after the game, second - of player2.

Ratings are computed based only on games between players of interest (see Players) and `NA` values.

Initial ratings should be defined with argument `initial_ratings`. It can be:

- A single numeric value. In this case initial ratings for all players are set to this value.
- A named vector of ratings. All non-`NA` players, for which rating is computed, should be present in its names (as character representation of players' actual identifiers).
- A data frame with first column representing player and second - initial rating. It will be converted to named vector with deframe() from `tibble` package.

Value

`rate_iterative()` returns a tibble with columns `player` (player identifier) and `rating_iterative` (Iterative ratings, based on row order, by the end of competition results). **Interpretation of numbers depends on rating function** `rate_fun`.

`rank_iterative()` returns a `tibble` with columns `player`, `rating_iterative` (if `keep_rating = TRUE`) and `ranking_iterative` (Iterative ranking computed with `round_rank()` based on specified `type`).

`add_iterative_ratings()` returns a widecr form of `cr_data` with four rating columns added:

- **rating1Before** - Rating of player1 before the game.
- **rating2Before** - Rating of player2 before the game.
- **rating1After** - Rating of player1 after the game.
- **rating2After** - Rating of player2 after the game.

Players

`comperank` offers a possibility to handle certain set of players. It is done by having `player` column (in longcr format) as factor with levels specifying all players of interest. In case of factor the result is returned only for players from its levels. Otherwise - for all present players.

Examples

```
test_rate_fun <- function(rating1, score1, rating2, score2) {
  c(rating1, rating2) + ((score1 >= score2) * 2 - 1) * c(1, -1)
}
set.seed(1002)
cr_data <- data.frame(
  game = rep(1:10, each = 2),
  player = rep(1:5, times = 4),
  score = runif(20)
)
cr_data$player[c(6, 8)] <- NA

# Different settings of add_iterative_ratings
add_iterative_ratings(cr_data, test_rate_fun)

add_iterative_ratings(cr_data, test_rate_fun, initial_ratings = 10)

add_iterative_ratings(
  cr_data, test_rate_fun,
  initial_ratings = c("1" = 1, "2" = 2, "3" = 3, "4" = 4, "5" = 5)
)

add_iterative_ratings(
  cr_data, test_rate_fun,
  initial_ratings = data.frame(1:5, 0:4)
)

# Ratings and ranking at the end of competition results.
```

```
rate_iterative(cr_data, test_rate_fun)

rank_iterative(cr_data, test_rate_fun, type = "desc")

rank_iterative(cr_data, test_rate_fun, type = "desc", keep_rating = TRUE)
```

keener — *Keener method*

Description

Functions to compute rating and ranking using Keener method.

Usage

```
rate_keener(cr_data, ..., fill = 0, force_nonneg_h2h = TRUE,
  skew_fun = skew_keener, normalize_fun = normalize_keener, eps = 0.001)

rank_keener(cr_data, ..., fill = 0, force_nonneg_h2h = TRUE,
  skew_fun = skew_keener, normalize_fun = normalize_keener, eps = 0.001,
  keep_rating = FALSE, ties = c("average", "first", "last", "random", "max",
  "min"), round_digits = 7)

skew_keener(x)

normalize_keener(mat, cr_data)
```

Arguments

`cr_data`	Competition results in format ready for as_longcr().
`...`	Head-to-Head expression (see h2h_mat()).
`fill`	A single value to use instead of NA for missing pairs.
`force_nonneg_h2h`	Whether to force nonnegative values in Head-to-Head matrix.
`skew_fun`	Skew function.
`normalize_fun`	Normalization function.
`eps`	Coefficient for forcing irreducibility.
`keep_rating`	Whether to keep rating column in ranking output.
`ties`	Value for `ties` in `round_rank()`.
`round_digits`	Value for `round_digits` in `round_rank()`.
`x`	Argument for `skew_keener()`.
`mat`	Argument for `normalize_keener()`.

Details

Keener rating method is based on Head-to-Head matrix of the competition results. Therefore it can be used for competitions with variable number of players. Its algorithm is as follows:

1. Compute Head-to-Head matrix of competition results based on Head-to-Head expression supplied in `...` (see h2h_mat() for technical details and section **Design of Head-to-Head values** for design details). Head-to-Head values are computed based only on the games between players of interest (see Players). Ensure that there are no `NA`s by using `fill` argument. If `force_nonneg_h2h` is `TRUE` then the minimum value is subtracted (in case some Head-to-Head value is strictly negative).
2. Update raw Head-to-Head values (denoted as S) with the pair-normalization: a_ij = (S_ij + 1) / (S_ij + S_ji + 2). This step should make comparing different players more reasonable.
3. Skew Head-to-Head values with applying `skew_fun` to them. `skew_fun` should take numeric vector as only argument. It should return skewed vector. The default skew function is `skew_keener()`. This step should make abnormal results not very abnormal. To omit this step supply `skew_fun = NULL`.
4. Normalize Head-to-Head values with `normalize_fun` using `cr_data`. `normalize_fun` should take Head-to-Head matrix as the first argument and `cr_data` as second. It should return normalized matrix. The default normalization is `normalize_keener()` which divides Head-to-Head value of 'player1'-'player2' matchup divided by the number of games played by 'player1' (error is thrown if there are no games). This step should take into account possibly not equal number of games played by players. To omit this step supply `normalize_keener = NULL`.
5. Add small value to Head-to-Head matrix to ensure its irreducibility. If all values are strictly positive then this step is omitted. In other case small value is computed as the smallest non-zero Head-to-Head value multiplied by `eps`. This step is done to ensure applicability of Perron-Frobenius theorem.
6. Compute Perron-Frobenius vector of the resultant matrix, i.e. the strictly positive real eigenvector (which values sum to 1) for eigenvalue (which is real) of the maximum absolute value. This vector is Keener rating vector.

If using `normalize_keener()` in normalization step, ensure to analyze players which actually played games (as division by a number of played games is made). If some player didn't play any game, en error is thrown.

Value

`rate_keener()` returns a tibble with columns `player` (player identifier) and `rating_keener` (Keener rating). Sum of all ratings should be equal to 1. **Bigger value indicates better player performance**.

`rank_keener()` returns a `tibble` with columns `player`, `rating_keener` (if `keep_rating = TRUE`) and `ranking_keener` (Keener ranking computed with `round_rank()`).

`skew_keener()` returns skewed vector of the same length as `x`.

`normalize_keener()` returns normalized matrix with the same dimensions as `mat`.

Design of Head-to-Head values

Head-to-Head values in these functions are assumed to follow the property which can be *equivalently* described in two ways:

- In terms of matrix format: **the more Head-to-Head value in row *i* and column *j* the better player from row *i* performed than player from column *j***.
- In terms of long format: **the more Head-to-Head value the better player1 performed than player2**.

This design is chosen because in most competitions the goal is to score **more points** and not less. Also it allows for more smooth use of h2h_funs from `comperes` package.

Players

`comperank` offers a possibility to handle certain set of players. It is done by having `player` column (in longcr format) as factor with levels specifying all players of interest. In case of factor the result is returned only for players from its levels. Otherwise - for all present players.

References

James P. Keener (1993) *The Perron-Frobenius theorem and the ranking of football teams*. SIAM Review, 35(1):80–93, 1993.

Examples

```
rate_keener(ncaa2005, sum(score1))

rank_keener(ncaa2005, sum(score1))

rank_keener(ncaa2005, sum(score1), keep_rating = TRUE)

# Impact of skewing
rate_keener(ncaa2005, sum(score1), skew_fun = NULL)

# Impact of normalization.
rate_keener(ncaa2005[-(1:2), ], sum(score1))

rate_keener(ncaa2005[-(1:2), ], sum(score1), normalize_fun = NULL)
```

`markov` *Markov method*

Description

Functions to compute rating and ranking using Markov method.

Usage

```
rate_markov(cr_data, ..., fill = list(), stoch_modify = teleport(0.15),
  weights = 1, force_nonneg_h2h = TRUE)

rank_markov(cr_data, ..., fill = list(), stoch_modify = teleport(0.15),
  weights = 1, force_nonneg_h2h = TRUE, keep_rating = FALSE,
  ties = c("average", "first", "last", "random", "max", "min"),
  round_digits = 7)
```

Arguments

`cr_data`	Competition results in format ready for as_longcr().
`...`	Name-value pairs of Head-to-Head functions (see h2h_long()).
`fill`	A named list that for each Head-to-Head function supplies a single value to use instead of NA for missing pairs (see h2h_long()).
`stoch_modify`	A single function to modify column-stochastic matrix or a list of them (see Stochastic matrix modifiers).
`weights`	Weights for different stochastic matrices.
`force_nonneg_h2h`	Whether to force nonnegative values in Head-to-Head matrix.
`keep_rating`	Whether to keep rating column in ranking output.
`ties`	Value for `ties` in `round_rank()`.
`round_digits`	Value for `round_digits` in `round_rank()`.

Details

Markov ratings are based on players 'voting' for other players being better. Algorithm is as follows:

1. 'Voting' is done with Head-to-Head values supplied in `...` (see h2h_mat() for technical details and section **Design of Head-to-Head values** for design details). Take special care of Head-to-Head values for self plays (when player1 equals player2). **Note** that Head-to-Head values should be non-negative. Use `force_nonneg_h2h = TRUE` to force that by subtracting minimum Head-to-Head value (in case some Head-to-Head value is strictly negative).
2. Head-to-Head values are transformed into matrix which is normalized to be column-stochastic (sum of every column should be equal to 1) Markov matrix *S*. **Note** that all missing values are converted into 0. To specify other value use `fill` argument.
3. *S* is modified with `stoch_modify` to deal with possible problems behind *S*, such as reducibility and rows with all 0.
4. Stationary vector is computed based on *S* as probability transition matrix of Markov chain process (transition probabilities from state **i** are elements from column **i**). The result is declared as Markov ratings.

Considering common values and structure of stochastic matrices one can naturally combine different 'votings' in one stochastic matrix:

1. Long format of Head-to-Head values is computed using `...` (which in this case should be several expressions for Head-to-Head functions).

2. Each set of Head-to-Head values is transformed into matrix which is normalized to column-stochastic.
3. Each stochastic matrix is modified with respective modifier which is stored in `stoch_modify` (which can be a list of functions).
4. The resulting stochastic matrix is computed as weighted average of modified stochastic matrices.

For Head-to-Head functions in `...` (considered as list) and argument `stoch_modify` general R recycling rule is applied. If `stoch_modify` is a function it is transformed to list with one function.

`weights` is recycled to the maximum length of two mentioned recycled elements and then is normalized to sum to 1.

Ratings are computed based only on games between players of interest (see Players).

Value

`rate_markov()` returns a tibble with columns `player` (player identifier) and `rating_markov` (Markov rating). The sum of all ratings should be equal to 1. **Bigger value indicates better player performance**.

`rank_markov` returns a `tibble` with columns `player`, `rating_markov` (if `keep_rating = TRUE`) and `ranking_markov` (Markov ranking computed with `round_rank()`).

Design of Head-to-Head values

Head-to-Head values in these functions are assumed to follow the property which can be *equivalently* described in two ways:

- In terms of matrix format: **the more Head-to-Head value in row *i* and column *j* the better player from row *i* performed than player from column *j***.
- In terms of long format: **the more Head-to-Head value the better player1 performed than player2**.

This design is chosen because in most competitions the goal is to score **more points** and not less. Also it allows for more smooth use of h2h_funs from `comperes` package.

Players

`comperank` offers a possibility to handle certain set of players. It is done by having `player` column (in longcr format) as factor with levels specifying all players of interest. In case of factor the result is returned only for players from its levels. Otherwise - for all present players.

References

Wikipedia page for Markov chain.

Examples

```
rate_markov(
  cr_data = ncaa2005,
  # player2 "votes" for player1 if player1 won
  comperes::num_wins(score1, score2, half_for_draw = FALSE),
  stoch_modify = vote_equal
)

rank_markov(
  cr_data = ncaa2005,
  comperes::num_wins(score1, score2, half_for_draw = FALSE),
  stoch_modify = vote_equal
)

rank_markov(
  cr_data = ncaa2005,
  comperes::num_wins(score1, score2, half_for_draw = FALSE),
  stoch_modify = vote_equal,
  keep_rating = TRUE
)

# Combine multiple stochastic matrices and
# use inappropriate `fill` which misrepresents reality
rate_markov(
  cr_data = ncaa2005[-(1:2), ],
  win = comperes::num_wins(score1, score2, half_for_draw = FALSE),
  # player2 "votes" for player1 proportionally to the amount player1 scored
  # more in direct confrontations
  score_diff = max(mean(score1 - score2), 0),
  fill = list(win = 0.5, score_diff = 10),
  stoch_modify = list(vote_equal, teleport(0.15)),
  weights = c(0.8, 0.2)
)
```

`massey` *Massey method*

Description

Functions to compute rating and ranking using Massey method.

Usage

```
rate_massey(cr_data)

rank_massey(cr_data, keep_rating = FALSE, ties = c("average", "first",
  "last", "random", "max", "min"), round_digits = 7)
```

Arguments

`cr_data`	Competition results in format ready for as_longcr().
`keep_rating`	Whether to keep rating column in ranking output.
`ties`	Value for `ties` in `round_rank()`.
`round_digits`	Value for `round_digits` in `round_rank()`.

Details

This rating method was initially designed for games between two players. There will be an error if in `cr_data` there is a game not between two players. Convert input competition results manually or with to_pairgames() from `comperes` package.

It is assumed that score is numeric and higher values are better for the player.

Computation is done based only on the games between players of interest (see Players). **Note** that all those players should be present in `cr_data` because otherwise there will be an error during solving linear system described below. Message is given if there are players absent in `cr_data`.

The outline of Massey rating method is as follows:

1. Compute Massey matrix: diagonal elements are equal to number of games played by certain player, off-diagonal are equal to minus number of common games played. This matrix will be the matrix of system of linear equations (SLE).
2. Compute right-hand side of SLE as cumulative score differences of players, i.e. sum of all scores *for* the player minus sum of all scores *against* the player.
3. Make adjustment for solvability of SLE. Modify the last row of Massey matrix so that all its cells are equal to 1. Also change the last cell in right-hand side to 0. This adjustment ensures that sum of all outcome ratings will be 0.
4. Solve the SLE. The solution is the Massey rating.

Value

`rate_massey()` returns a tibble with columns `player` (player identifier) and `rating_massey` (Massey rating). The sum of all ratings should be equal to 0. **Bigger value indicates better player performance**.

`rank_massey()` returns a `tibble` with columns `player`, `rating_massey` (if `keep_rating = TRUE`) and `ranking_massey` (Massey ranking computed with `round_rank()`).

Players

`comperank` offers a possibility to handle certain set of players. It is done by having `player` column (in longcr format) as factor with levels specifying all players of interest. In case of factor the result is returned only for players from its levels. Otherwise - for all present players.

References

Kenneth Massey (1997) *Statistical models applied to the rating of sports teams*. Bachelor's thesis, Bluefield College.

Examples

```
rate_massey(ncaa2005)

rank_massey(ncaa2005)

rank_massey(ncaa2005, keep_rating = TRUE)
```

`offense-defense` *Offense-Defense method*

Description

Functions to compute rating and ranking using Offense-Defense method.

Usage

```
rate_od(cr_data, ..., force_nonneg_h2h = TRUE, eps = 0.001, tol = 1e-04,
  max_iterations = 100)

rank_od(cr_data, ..., force_nonneg_h2h = TRUE, eps = 0.001, tol = 1e-04,
  max_iterations = 100, keep_rating = FALSE, ties = c("average", "first",
  "last", "random", "max", "min"), round_digits = 7)
```

Arguments

`cr_data`	Competition results in format ready for as_longcr().
`...`	Head-to-Head expression (see h2h_mat()).
`force_nonneg_h2h`	Whether to force nonnegative values in Head-to-Head matrix.
`eps`	Coefficient for total support.
`tol`	Tolerance value for iterative algorithm.
`max_iterations`	Maximum number of iterations for iterative algorithm.
`keep_rating`	Whether to keep rating columns in ranking output.
`ties`	Value for `ties` in `round_rank()`.
`round_digits`	Value for `round_digits` in `round_rank()`.

Details

Offense-Defense (OD) rating is designed for games in which player's goal is to make higher score than opponent(s). To describe competition results Head-to-Head matrix is computed using ... (see h2h_mat() for technical details and section **Design of Head-to-Head values** for design details). For pairs of players without common games Head-to-Head value is computed to 0 (not `NA`). **Note** that values should be non-negative and non-NA. This can be ensured with setting `force_nonneg_h2h` to `TRUE`.

For player which can achieve *high* Head-to-Head value (even against the player with strong defense) it is said that he/she has **strong offense** which results into *high* offensive rating. For player which can force their opponents into achieving *low* Head-to-Head value (even if they have strong offense) it is said that he/she has **strong defense** which results into *low* defensive rating.

Offensive and defensive ratings describe different skills of players. In order to fully rate players, OD ratings are computed: offensive ratings divided by defensive. The more OD rating the better player performance.

Algorithm for OD ratings is as follows:

1. Compute Head-to-Head matrix using `...`.
2. Add small value to Head-to-Head matrix to ensure convergence of the iterative algorithm in the next step. If all values are strictly positive then this step is omitted. In other case small value is computed as the smallest non-zero Head-to-Head value multiplied by `eps`.
3. Perform iterative fixed point search with the following recurrent formula: `def_{k+1} = t(A) %*% inv(A %*% inv(def_{k}))` where `def_{k}` is a vector of defensive ratings at iteration `k`, `A` is a perturbed Head-to-Head matrix, `inv(x) = 1 / x`. Iterative search stops if at least one of two conditions is met:
   - `sum(abs(def_{k+1} / def_{k} - 1)) < tol`.
   - Number of iterations exceeds maximum number of iterations `max_iterations`.
4. Compute offensive ratings: `off = A %*% inv(def)`.
5. Compute OD ratings: `od = off / def`.

Ratings are computed based only on games between players of interest (see Players). However, be careful with OD ratings for players with no games: they will have weak offense (because they "scored" 0 in all games) but strong defense (because all their opponents also "scored" 0 in all common games). Therefore accounting for missing players might be not a very good idea.

Value

`rate_od()` returns a tibble with the following columns:

- **player** - player identifier.
- **rating_off** - offensive rating of player. **Bigger value indicates better player performance**.
- **rating_def** - defensive rating of player. **Smaller value indicates better player performance**.
- **rating_od** - Offense-Defense rating of player. **Bigger value indicates better player performance**.

`rank_od()` returns a `tibble` of the similar structure as `rate_od()`:

- **player** - player identifier.
- **rating_off**, **rating_def**, **rating_od** - ratings (if `keep_rating = TRUE`).
- **ranking_off**, **ranking_def**, **ranking_od** - rankings computed with `round_rank()`.

Design of Head-to-Head values

Head-to-Head values in these functions are assumed to follow the property which can be *equivalently* described in two ways:

- In terms of matrix format: **the more Head-to-Head value in row *i* and column *j* the better player from row *i* performed than player from column *j***.
- In terms of long format: **the more Head-to-Head value the better player1 performed than player2**.

This design is chosen because in most competitions the goal is to score **more points** and not less. Also it allows for more smooth use of h2h_funs from `comperes` package.

Players

`comperank` offers a possibility to handle certain set of players. It is done by having `player` column (in longcr format) as factor with levels specifying all players of interest. In case of factor the result is returned only for players from its levels. Otherwise - for all present players.

References

Amy N. Langville, Carl D. Meyer (2012) *Who's #1?: The science of rating and ranking*.

Philip A. Knight (2008) *The Sinkhorn-Knopp algorithm: Convergence and applications.*. SIAM Journal of Matrix Analysis, 30(1):261–275, 2008 (For stopping rule of iterative algorithm).

Examples

```
rate_od(ncaa2005, mean(score1))

rank_od(ncaa2005, mean(score1))

rank_od(ncaa2005, mean(score1), keep_rating = TRUE)

# Account for self play
rate_od(ncaa2005, if(player1[1] == player2[1]) 0 else mean(score1))
```

`rating-ranking` *Definition of Rating and Ranking*

Description

- **Rating** - List (in the ordinary sense) of numerical values, one for each player, or the numerical value itself. Its interpretation depends on rating method: either bigger value indicates better player performance or otherwise.
- **Ranking** - Rank-ordered list (in the ordinary sense) of players: rank 1 indicates player with best performance.

round_rank *Rank vector after rounding*

Description

Function for ranking vector after rounding.

Usage

```
round_rank(x, type = "desc", na.last = TRUE, ties = c("average", "first",
  "last", "random", "max", "min"), round_digits = 7)
```

Arguments

x	A numeric, complex, character or logical vector.
type	Type of ranking: "desc" or "asc" (see Details).
na.last	For controlling the treatment of NAs. If TRUE, missing values in the data are put last; if FALSE, they are put first; if NA, they are removed; if "keep" they are kept with rank NA.
ties	A character string specifying how ties are treated (see Details). Can be abbreviated.
round_digits	Value of digits for round().

Details

This is basically a wrapper around rank() in which x is pre-modified by rounding to specific number of digits round_digits.

type can have two values: "desc" for ranking in descending order (rank 1 is given to the biggest value in x) and "asc" (rank 1 is given to the smallest value in x). Any other value will cause error.

Value

A numeric vector of the same length as x with names copied from x (unless na.last = NA, when missing values are removed). The vector is of integer type unless x is a long vector or ties = "average" when it is of double type (whether or not there are any ties).

Examples

```
round_rank(10:1, type = "desc")
round_rank(10:1, type = "asc")

set.seed(334)
x <- 10^(-10) * runif(10)
round_rank(x)
```

snooker_events *Snooker events*

Description

Data set describing snooker events in seasons 2016/2017 and 2017/2018.

Usage

```
snooker_events
```

Format

A tibble with one row per event and the following columns:

- **id** `<int>` : Event identifier in snooker.org database (used in `eventId` column of snooker_matches).
- **season** `<int>` : Season identifier (by the year of season start).
- **name** `<chr>` : Event name.
- **startDate** `<dttm>` : Start date of event.
- **endDate** `<dttm>` : End date of event.
- **sponsor** `<chr>` : Event sponsor name.
- **type** `<chr>` : Event type ("Invitational", "Qualifying", or "Ranking").
- **venue** `<chr>` : Venue name event was played.
- **city** `<chr>` : City name event was played.
- **country** `<chr>` : Country name event was played.

Details

Data is taken from snooker.org (http://www.snooker.org/) API.

This data set has information about events that have all following qualities:

- It has "Invitational", "Qualifying", or "Ranking" type.
- It describes traditional snooker (not 6 Red or Power Snooker) between individual players (not teams).
- Both genders can take part (not only men or women).
- Players of all ages can take part (not only seniors or under 21).
- It is not "Shoot-Out" as those events are treated differently in snooker.org database.

See Also

Snooker players, snooker matches

`snooker_matches` *Snooker matches*

Description

Data set describing snooker matches in seasons 2016/2017 and 2017/2018.

Usage

```
snooker_matches
```

Format

A tibble with one row per match and the following columns:

- **id** `<int>` : Match identifier in snooker.org database.
- **eventId** `<int>` : Match's event identifier (taken from `id` column of snooker_events)
- **round** `<int>` : Round number of event in which match was played. *Usually* event's structure is organized in rounds: sets of matches with roughly "the same importance". *Usually* the more round number the "more important" matches are played. However, there are many exceptions.
- **player1Id** `<int>` : Identifier of first player in match (taken from `id` column of snooker_players).
- **score1** `<int>` : Number of won frames (individual games) by first player.
- **walkover1** `<lgl>` : Whether the win of first player was scored by the technical reasons.
- **player2Id** `<int>` : Identifier of second player in match (taken from `id` column of snooker_players).
- **score2** `<int>` : Number of won frames (individual games) by second player.
- **walkover2** `<lgl>` : Whether the win of second player was scored by the technical reasons.
- **winnerId** `<int>` : Identifier of match's winner (taken from either `player1Id` or `player2Id` columns).
- **startDate** `<dttm>` : Time at which match started.
- **endDate** `<dttm>` : Time at which match ended.
- **scheduledDate** `<dttm>` : Time at which match was scheduled to start.
- **frameScores** `<chr>` : Scores of players in frames. Usually is missing, present only for important matches.

Details

Data is taken from snooker.org (http://www.snooker.org/) API.

Matches are present only for tracked snooker events.

See Also

Snooker events, snooker players

`snooker_players` *Snooker players*

Description

Data set describing snooker players in seasons 2016/2017 and 2017/2018.

Usage

```
snooker_players
```

Format

A tibble with one row per player and the following columns:

- **id** `<int>` : Player identifier in snooker.org database (used in `player1Id`, `player2Id` and `winnerId` columns of snooker_matches).
- **name** `<chr>` : Player full name.
- **nationality** `<chr>` : Player nationality.
- **sex** `<chr>` : Player gender ("F" for female, "M" for male, and "Unknown").
- **born** `<dttm>` : Player date of birth.
- **status** `<chr>` : Player status in season 2017/2018 ("pro" for professional, "ama" for amateur).

Details

Data is taken from snooker.org (http://www.snooker.org/) API.

Data is present only for players who played at least one game in tracked snooker events in seasons 2016/2017 and 2017/2018.

See Also

Snooker events, snooker matches

`stoch-modifiers` *Stochastic matrix modifiers*

Description

Functions for stochastic matrix modifications.

Usage

```
teleport(teleport_prob = 0.15)

vote_equal(stoch)

vote_self(stoch)
```

Arguments

`teleport_prob`	Probability of 'teleportation'.
`stoch`	Input stochastic matrix.

Details

Modification logic behind `teleport()` assumes that at each step of Markov chain (described by column-stochastic matrix) the decision is made whether to change state according to stochastic matrix or to 'teleport' to any state with equal probability. Probability of 'teleport' is `teleport_prob`. This modification is useful because it ensures irreducibility of stochastic matrix (with `teleport_prob` in (0; 1)). **Note** that in order to obtain modifier one should call function `teleport()` with some parameter.

`vote_equal()` and `vote_self()` modify columns with elements only equal to 0. The former fills them with `1/nrow(stoch)` and the latter changes only the respective diagonal element to 1. This is equivalent to jump to any state with equal probability and to stay in the current state respectively.

Value

`teleport()` returns a modifier function.

`vote_equal()` and `vote_self()` are modifier functions and return modified version of input stochastic matrix.

Examples

```
input_stoch <- matrix(c(0, 0.3,
                        0, 0.7),
                      ncol = 2, byrow = TRUE)
teleport(0.15)(input_stoch)

vote_equal(input_stoch)

vote_self(input_stoch)
```

Index