Package 'pcpr'

July 23, 2025

Type Package

Title Principal Component Pursuit for Environmental Epidemiology

Version 1.0.0

Description Implementation of the pattern recognition technique Principal Component Pursuit tailored to environmental health data, as described in Gibson et al (2022) <doi:10.1289/EHP10479>.

License GPL (>= 3)

URL https://columbia-prime.github.io/pcpr/, https://github.com/Columbia-PRIME/pcpr

BugReports https://github.com/Columbia-PRIME/pcpr/issues

Depends R (>= 3.5.0)

Imports checkmate, dplyr, foreach, furrr, future, magrittr, progressr, purrr, rlang, tidyselect

Suggests GGally, ggplot2, knitr, lubridate, metR, rmarkdown, tidyr

VignetteBuilder knitr

Encoding UTF-8

LazyData true

RoxygenNote 7.3.2

NeedsCompilation no

Author Lawrence G. Chillrud [aut, cre, cph] (ORCID: <https://orcid.org/0000-0003-0727-0161>), Jaime Benavides [aut] (ORCID: <https://orcid.org/0000-0002-1851-5155>), Elizabeth A. Gibson [aut] (ORCID: <https://orcid.org/0000-0001-5119-5133>), Junhui Zhang [aut] (ORCID: <https://orcid.org/0009-0008-5922-1058>), Jingkai Yan [aut] (ORCID: <https://orcid.org/0000-0002-2094-2092>), John N. Wright [aut], Jeff Goldsmith [aut], Marianthi-Anna Kioumourtzoglou [aut] (ORCID: <https://orcid.org/0000-0001-5710-4992>), Columbia University [fnd] (ROR: <https://ror.org/00hj8s172>)

Maintainer Lawrence G. Chillrud <lawrencechillrud@gmail.com>

Repository CRAN

Date/Publication 2025-03-27 18:20:02 UTC

Contents

`get_pcp_defaults` *Retrieve default PCP parameter settings for given matrix*

Description

`get_pcp_defaults()` calculates "default" PCP parameter settings `lambda`, `mu` (used in `root_pcp()`), and `eta` (used in `rrmc()`) for a given data matrix `D`.

The "default" values of `lambda` and `mu` offer *theoretical* guarantees of optimal estimation performance. Candès et al. (2011) obtained the guarantee for `lambda`, while Zhang et al. (2021) obtained the result for `mu`. *It has not yet been proven whether or not* `eta` *enjoys similar properties.*

In practice it is common to find different optimal parameter values after tuning these parameters in a grid search. Therefore, **it is recommended to use these defaults primarily to help define a reasonable initial parameter search space to pass into `grid_search_cv()`.**

Usage

```
get_pcp_defaults(D)
```

Arguments

`D`	The input data matrix.

Value

A list containing:

- `lambda`: The theoretically optimal `lambda` value used in `root_pcp()`.
- `mu`: The theoretically optimal `mu` value used in `root_pcp()`.
- `eta`: The default `eta` value used in `rrmc()`.

The intuition behind PCP parameters

`root_pcp()`'s objective function is given by:

$$\min_{L,S} ||L||_* + \lambda||S||_1 + \mu||L + S - D||_F$$

- `lambda` controls the sparsity of `root_pcp()`'s output `S` matrix; larger values of `lambda` penalize non-zero entries in `S` more stringently, driving the recovery of sparser `S` matrices. Therefore, if you a priori expect few outlying events in your model, you might expect a grid search to recover relatively larger `lambda` values, and vice-versa.
- `mu` adjusts `root_pcp()`'s sensitivity to noise; larger values of `mu` penalize errors between the predicted model and the observed data (i.e. noise), more severely. Environmental data subject to higher noise levels therefore require a `root_pcp()` model equipped with smaller `mu` values (since higher noise means a greater discrepancy between the observed mixture and the true underlying low-rank and sparse model). In virtually noise-free settings (e.g. simulations), larger values of `mu` would be appropriate.

`rrmc()`'s objective function is given by:

$$\min_{L,S} I_{rank(L)\leq r} + \eta||S||_0 + ||L + S - D||_F^2$$

- `eta` controls the sparsity of `rrmc()`'s output `S` matrix, just as `lambda` does for `root_pcp()`. Because there are no other parameters scaling the noise term, `eta` can be thought of as a ratio between `root_pcp()`'s `lambda` and `mu`: Larger values of `eta` will place a greater emphasis on penalizing the non-zero entries in `S` over penalizing the errors between the predicted and observed data (the dense noise `Z`).

The calculation of the "default" PCP parameters

- `lambda` is calculated as $\lambda = 1/\sqrt{\max(n, p)}$, where n and p are the dimensions of the input matrix $D_{n\times p}$ Candès et al. (2011).
- `mu` is calculated as $\mu = \sqrt{\frac{\min(n,p)}{2}}$, where n and p are as above [Zhang et al. (2021)].
- `eta` is simply $\eta = \frac{\lambda}{\mu}$.

References

Candès, Emmanuel J., Xiaodong Li, Yi Ma, and John Wright. "Robust principal component analysis?." Journal of the ACM (JACM) 58, no. 3 (2011): 1-37.

Zhang, Junhui, Jingkai Yan, and John Wright. "Square root principal component pursuit: tuning-free noisy robust matrix recovery." Advances in Neural Information Processing Systems 34 (2021): 29464-29475. [available here]

See Also

grid_search_cv()

Examples

```
# Examine the queens PM2.5 data
queens
# Get rid of the Date column
D <- as.matrix(queens[, 2:ncol(queens)])
# Get default PCP parameters
default_params <- get_pcp_defaults(D)
# Use default parameters to define parameter search space
scaling_factors <- sort(c(10^seq(-2, 4, 1), 2 * 10^seq(-2, 4, 1)))
etas_to_grid_search <- default_params$eta * scaling_factors
etas_to_grid_search
```

---

grid_search_cv *Cross-validated grid search for PCP models*

Description

grid_search_cv() conducts a Monte Carlo style cross-validated grid search of PCP parameters for a given data matrix D, PCP function pcp_fn, and grid of parameter settings to search through grid. The run time of the grid search can be sped up using bespoke parallelization settings. The call to grid_search_cv() can be wrapped in a call to progressr::with_progress() for progress bar updates. See the below sections for details.

Usage

```
grid_search_cv(
  D,
  pcp_fn,
  grid,
  ...,
  parallel_strategy = "sequential",
  num_workers = 1,
  perc_test = 0.05,
  num_runs = 100,
  return_all_tests = FALSE,
  verbose = TRUE
)
```

Arguments

D	The input data matrix (can contain NA values). Note that PCP will converge much more quickly when D has been standardized in some way (e.g. scaling columns by their standard deviations, or column-wise min-max normalization).

pcp_fn
: The PCP function to use when grid searching. Must be either rrmc or root_pcp (passed without the soft brackets).

grid
: A data.frame of dimension j by k containing the j-many unique settings of k-many parameters to try. **NOTE: The columns of grid should be named after the required parameters in the function header of pcp_fn.** For example, if pcp_fn = root_pcp and you want to search through lambda and mu, then names(grid) must be set to c("lambda", "mu"). If instead you want to keep e.g. lambda fixed and search through only mu, you can either have a grid with only one column, mu, and pass lambda as a constant via ..., or you can have names(grid) set to c("lambda", "mu") where lambda is constant. The same logic applies for pcp_fn = rrmc and eta and r.

...
: Any parameters required by pcp_fn that should be kept constant throughout the grid search, or those parameters that cannot be stored in grid (e.g. the LOD parameter). A parameter should not be passed with ... if it is already a column in grid, as that behavior is ambiguous.

parallel_strategy
: (Optional) The parallelization strategy used when conducting the grid search (to be passed on to the future::plan() function). Must be one of: "sequential", "multisession", "multicore" or "cluster". By default, parallel_strategy = "sequential", which runs the grid search in serial and the num_workers argument is ignored. The option parallel_strategy = "multisession" parallelizes the search via sockets in separate R *sessions*. The option parallel_strategy = "multicore" is not supported on Windows machines, nor in .Rmd files (must be run in a .R script) but parallelizes the search much faster than "multisession" since it runs separate *forked* R processes. The option parallel_strategy = "cluster" parallelizes using separate R sessions running typically on one or more machines. Support for other parallel strategies will be added in a future release of pcpr. **It is recommended to use** parallel_strategy = "multicore" **or** "multisession" **when possible.**

num_workers
: (Optional) An integer specifying the number of workers to use when parallelizing the grid search, to be passed on to future::plan(). By default, num_workers = 1. When possible, it is recommended to use num_workers = parallel::detectCores(logical = F), which computes the number of physical CPUs available on the machine (see parallel::detectCores()). num_workers is ignored when parallel_strategy = "sequential", and must be > 1 otherwise.

perc_test
: (Optional) The fraction of entries of D that will be randomly corrupted as NA missing values (the test set). Can be anthing in the range [0, 1). By default, perc_test = 0.05. See **Best practices** section for more details.

num_runs
: (Optional) The number of times to test a given parameter setting. By default, num_runs = 100. See **Best practices** section for more details.

return_all_tests
: (Optional) A logical indicating if you would like the output from all the calls made to pcp_fn over the course of the grid search to be returned to you in list format. If set to FALSE, then only statistics on the parameters tested will be returned. If set to TRUE then every L, and S matrix recovered during the grid search will be returned in the lists L_mats and S_mats, every test set matrix will be returned in the list test_mats, the original input matrix will be

returned as original_mat, and the parameters passed in to ... will be returned in the constant_params list. **By default,** return_all_tests = FALSE**, which is highly recommended. Setting** return_all_tests = TRUE **can consume a massive amount of memory depending on the size of** grid**, the input matrix** D**, and the value for** num_runs**.**

verbose — (Optional) A logical indicating if you would like verbose output displayed or not. By default, verbose = TRUE. To obtain progress bar updates, the user must wrap the grid_search_cv() call with a call to progressr::with_progress(). The progress bar does *not* depend on the value passed for verbose.

Value

A list containing:

- all_stats: A data.frame containing the statistics of every run comprising the grid search. These statistics include the parameter settings for the run, along with the run_num (used as the seed for the corruption step, step 1 in the grid search procedure), the relative error for the run rel_err, the rank of the recovered L matrix L_rank, the sparsity of the recovered S matrix S_sparsity, the number of iterations PCP took to reach convergence (for root_pcp() only), and the error status run_error of the PCP run (NA if no error, otherwise a character string).
- summary_stats: A data.frame containing a summary of the information in all_stats. Summary made by column-wise averaging the results in all_stats.
- metadata: A character string containing the metadata associated with the grid search instance.

If return_all_tests = TRUE then the following are also returned as part of the list:

- L_mats: A list containing all the L matrices returned from PCP throughout the grid search. Therefore, length(L_mats) == nrow(all_stats). Row i in all_stats corresponds to L_mats[[i]].
- S_mats: A list containing all the S matrices returned from PCP throughout the grid search. Therefore, length(S_mats) == nrow(all_stats). Row i in all_stats corresponds to S_mats[[i]].
- test_mats: A list of length(num_runs) containing all the corrupted test mats (and their masks) used throughout the grid search. Note: all_stats$run[i] corresponds to test_mats[[i]].
- original_mat: The original data matrix D.
- constant_params: A copy of the constant parameters that were originally passed to the grid search (for record keeping).

The Monte Carlo style cross-validation procedure

Each hyperparameter setting is cross-validated by:

1. Randomly corrupting perc_test percent of the entries in D as missing (i.e. NA values), yielding D_tilde. Done via sim_na().
2. Running the PCP function pcp_fn on D_tilde, yielding estimates L and S.
3. Recording the relative recovery error of L compared with the input data matrix D for *only those values that were imputed as missing during the corruption step* (step 1 above). Mathematically, calculate: $||P_{\Omega^c}(D - L)||_F / ||P_{\Omega^c}(D)||_F$, where P_{Ω^c} selects only those entries where is.na(D_tilde) == TRUE.

4. Repeating steps 1-3 for a total of num_runs-many times, where each "run" has a unique random seed from 1 to num_runs associated with it.
5. Performance statistics can then be calculated for each "run", and then summarized across all runs for average model performance statistics.

Best practices for perc_test and num_runs

Experimentally, this grid search procedure retrieves the best performing PCP parameter settings when perc_test is relatively low, e.g. perc_test = 0.05, or 5%, and num_runs is relatively high, e.g. num_runs = 100.

The larger perc_test is, the more the test set turns into a matrix completion problem, rather than the desired matrix decomposition problem. To better resemble the actual problem PCP will be faced with come inference time, perc_test should therefore be kept relatively low.

Choosing a reasonable value for num_runs is dependent on the need to keep perc_test relatively low. Ideally, a large enough num_runs is used so that many (if not all) of the entries in D are likely to eventually be tested. Note that since test set entries are chosen randomly for all runs 1 through num_runs, in the pathologically worst case scenario, the same exact test set could be drawn each time. In the best case scenario, a different test set is obtained each run, providing balanced coverage of D. Viewed another way, the smaller num_runs is, the more the results are susceptible to overfitting to the relatively few selected test sets.

Interpretaion of results

Once the grid search of has been conducted, the optimal hyperparameters can be chosen by examining the output statistics summary_stats. Below are a few suggestions for how to interpret the summary_stats table:

- Generally speaking, the first thing a user will want to inspect is the rel_err statistic, capturing the relative discrepancy between recovered test sets and their original, observed (yet possibly noisy) values. Lower rel_err means the PCP model was better able to recover the held-out test set. So, in general, **the best parameter settings are those with the lowest** rel_err**.** Having said this, it is important to remember that this statistic should be taken with a grain of salt: Because no ground truth L matrix exists, the rel_err measurement is forced to rely on the comparison between the *noisy observed data* matrix D and the *estimated low-rank model* L. So the rel_err metric is an "apples to oranges" relative error. For data that is a priori expected to be subject to a high degree of noise, it may actually be better to *discard* parameter settings with *suspiciously low* rel_errs (in which case the solution may be hallucinating an inaccurate low-rank structure from the observed noise).
- For grid searches using root_pcp() as the PCP model, parameters that fail to converge can be discarded. Generally, fewer root_pcp() iterations (num_iter) taken to reach convergence portend a more reliable / stable solution. In rare cases, the user may need to increase root_pcp()'s max_iter argument to reach convergence. rrmc() does not report convergence metadata, as its optimization scheme runs for a fixed number of iterations.
- Parameter settings with unreasonable sparsity or rank measurements can also be discarded. Here, "unreasonable" means these reported metrics flagrantly contradict prior assumptions, knowledge, or work. For instance, most air pollution datasets contain a number of extreme exposure events, so PCP solutions returning sparse S models with 100% sparsity have obviously been regularized too heavily. Solutions with lower sparsities should be preferred. Note

that reported sparsity and rank measurements are *estimates heavily dependent on the* `thresh` *set by the* `sparsity()` *&* `matrix_rank()` *functions*. E.g. it could be that the actual average matrix rank is much higher or lower when a threshold that better takes into account the relative scale of the singular values is used. Likewise for the sparsity estimations. Also, recall that the given value for `perc_test` artifically sets a sparsity floor, since those missing entries in the test set cannot be recovered in the `S` matrix. E.g. if `perc_test = 0.05`, then no parameter setting will have an estimated sparsity lower than 5%.

See Also

`sim_na()`, `sparsity()`, `matrix_rank()`, `get_pcp_defaults()`

Examples

```
#### -------Simple simulated PCP problem-------####
# First we will simulate a simple dataset with the sim_data() function.
# The dataset will be a 100x10 matrix comprised of:
# 1. A rank-3 component as the ground truth L matrix;
# 2. A ground truth sparse component S w/outliers along the diagonal; and
# 3. A dense Gaussian noise component
data <- sim_data()
#### -------Tiny grid search-------####
# Here is a tiny grid search just to test the function quickly.
# In practice we would recommend a larger grid search.
# For examples of larger searches, see the vignettes.
gs <- grid_search_cv(
  data$D,
  rrmc,
  data.frame("eta" = 0.35),
  r = 3,
  num_runs = 2
)
gs$summary_stats
```

`hard_threshold` *Hard-thresholding operator*

Description

`hard_threshold()` implements the hard-thresholding operator on a given matrix `D`, making `D` sparser: elements of `D` whose absolute value are less than a given threshold `thresh` are set to 0, i.e. $D[|D| < thresh] = 0$.

This is used in the non-convex PCP function `rrmc()` to provide a non-convex replacement for the `prox_l1()` method used in the convex PCP function `root_pcp()`. It is used to iteratively model the sparse `S` matrix with the help of an adaptive threshold (`thresh` changes over the course of optimization).

Usage

```
hard_threshold(D, thresh)
```

Arguments

D	The input data matrix.
thresh	The scalar-valued hard-threshold acting on D such that D[i, j] = 0 when abs(D[i, j]) < thresh, and D[i, j] = D[i, j] otherwise.

Value

The hard-thresholded matrix.

Examples

```
set.seed(42)
D <- matrix(rnorm(25), 5, 5)
S <- hard_threshold(D, thresh = 1)
D
S
```

---

impute_matrix *Impute missing values in given matrix*

Description

impute_matrix() imputes the missing NA values in a given matrix using a given imputation_scheme.

Usage

```
impute_matrix(D, imputation_scheme)
```

Arguments

D	The input data matrix.
imputation_scheme	The values to replace missing NA values in D with. Can be either: • A scalar numeric, indicating all NA values should be imputed with the same scalar numeric value; • A vector of length ncol(D), signifying column-specific imputation, where each entry in the imputation_scheme vector corresponds to the imputation value for each column in D; or • A matrix of dimension dim(D), indicating an observation-specific imputation scheme, where each entry in the imputation_scheme matrix corresponds to the imputation value for each entry in D.

Value

The imputed matrix.

See Also

sim_na(), sim_lod(), sim_data()

Examples

```
#### ------------Imputation with a scalar------------####
# simulate a small 5x5 mixture
D <- sim_data(5, 5)$D
# corrupt the mixture with 40% missing observations
D_tilde <- sim_na(D, 0.4)$D_tilde
D_tilde
# impute missing values with 0
impute_matrix(D_tilde, 0)
# impute missing values with -1
impute_matrix(D_tilde, -1)

#### ------------Imputation with a vector------------####
# impute missing values with the column-mean
impute_matrix(D_tilde, apply(D_tilde, 2, mean, na.rm = TRUE))
# impute missing values with the column-min
impute_matrix(D_tilde, apply(D_tilde, 2, min, na.rm = TRUE))

#### ------------Imputation with a matrix------------####
# impute missing values with random Gaussian noise
noise <- matrix(rnorm(prod(dim(D_tilde))), nrow(D_tilde), ncol(D_tilde))
impute_matrix(D_tilde, noise)

#### ------------Imputation with LOD/sqrt(2)------------####
D <- sim_data(5, 5)$D
lod_info <- sim_lod(D, q = 0.2)
D_tilde <- lod_info$D_tilde
D_tilde
lod <- lod_info$lod
impute_matrix(D_tilde, lod / sqrt(2))
```

`matrix_rank` *Estimate rank of a given matrix*

Description

`matrix_rank()` estimates the rank of a given data matrix `D` by counting the number of "practically nonzero" singular values of `D`.

The rank of a matrix is the number of linearly independent columns or rows in the matrix, governing the structure of the data. It can intuitively be thought of as the number of inherent latent patterns in the data.

A singular value s is determined to be "practically nonzero" if $s \geq s_{max} \cdot thresh$, i.e. if it is greater than or equal to the maximum singular value in `D` scaled by a given threshold `thresh`.

Usage

```
matrix_rank(D, thresh = NULL)
```

Arguments

D	The input data matrix (cannot have NA values).
thresh	(Optional) A double > 0, specifying the relative threshold by which "practically zero" is determined, used to calculate the rank of D. By default, thresh = NULL, in which case the threshold is set to max(dim(D)) * .Machine$double.eps.

Value

An integer estimating the rank of D.

See Also

sparsity()

Examples

```
data <- sim_data()
matrix_rank(data$D)
matrix_rank(data$L)
```

---

proj_rank_r *Project matrix to rank r*

Description

proj_rank_r() implements a best (i.e. closest) rank-r approximation of an input matrix.

This is computed via a simple truncated singular value decomposition (SVD), retaining the first r leading singular values/vectors of D. This is equivalent to solving the following optimization problem: $min||X - D||_F s.t. rank(X) <= r$, where X is the approximated solution and D is the input matrix.

proj_rank_r() is used to iteratively model the low-rank L matrix in the non-convex PCP function rrmc(), providing a non-convex replacement for the prox_nuclear() method used in the convex PCP function root_pcp().

Intuitively, proj_rank_r() can also be thought of as providing a PCA estimate of a rank-r matrix L from observed data D.

Usage

```
proj_rank_r(D, r)
```

Arguments

D	The input data matrix (cannot have NA values).
r	The rank that D should be projected/truncated to.

Value

The best rank-r approximation to D via a truncated SVD.

See Also

rrmc()

Examples

```
# Simulating a simple dataset D with the sim_data() function.
# The dataset will be a 10x5 matrix comprised of:
# 1. A rank-1 component as the ground truth L matrix; and
# 2. A dense Gaussian noise component corrupting L, making L full-rank
data <- sim_data(10, 5, 1, numeric(), 0.01)
# The observed matrix D is full-rank, while L is rank-1:
data.frame("D_rank" = matrix_rank(data$D), "L_rank" = matrix_rank(data$L))
before_proj_err <- norm(data$D - data$L, "F") / norm(data$L, "F")
# Projecting D onto the nearest rank-1 approximation, X, via proj_rank_r()
X <- proj_rank_r(data$D, r = 1)
after_proj_err <- norm(X - data$L, "F") / norm(data$L, "F")
proj_v_obs_err <- norm(X - data$D, "F") / norm(data$D, "F")
data.frame(
  "Observed_error" = before_proj_err,
  "Projected_error" = after_proj_err,
  "Projected_vs_observed_error" = proj_v_obs_err
)
```

---

queens *Daily chemical concentrations of 26 PM2.5 species from Queens, NYC (2001-2021)*

Description

A dataset containing the chemical concentrations (in µg/m^3) of 26 PM2.5 species measured every three to six days from 04/04/2001 through 12/30/2021 in Queens, New York City. Data obtained from the U.S. Environmental Protection Agency's Air Quality System data mart (site ID: 36-081-0124).

Usage

```
queens
```

Format

A tibble with 2443 rows and 27 variables:

- `Date`: The date the PM2.5 measurements were made
- `...`: The remaining 26 variables are the 26 PM2.5 species (in µg/m^3): Al, NH4, As, Ba, Br, Cd, Ca, Cl, Cr, Cu, EC, Fe, Pb, Mg, Mn, Ni, OC, K, Se, Si, Na, S, Ti, NO3, V, Zn

Source

https://epa.maps.arcgis.com/apps/webappviewer/index.html?id=5f239fd3e72f424f98ef3d5def547eb5

References

US Environmental Protection Agency. Air Quality System Data Mart internet database available via https://www.epa.gov/outdoor-air-quality-data. Accessed July 15, 2022.

Examples

```
queens
```

`root_pcp` *Square root principal component pursuit (convex PCP)*

Description

`root_pcp()` implements the convex PCP algorithm "Square root principal component pursuit" as described in Zhang et al. (2021) , outfitted with environmental health (EH)-specific extensions as described in Gibson et al. (2022).

Given an observed data matrix `D`, and regularization parameters `lambda` and `mu`, `root_pcp()` aims to find the best low-rank and sparse estimates `L` and `S`. The `L` matrix encodes latent patterns that govern the observed data. The `S` matrix captures any extreme events in the data unexplained by the underlying patterns in `L`.

Being convex, `root_pcp()` determines the rank `r`, or number of latent patterns in the data, autonomously during it's optimization. As such, the user does not need to specify the desired rank `r` of the output `L` matrix as in the non-convex PCP model `rrmc()`.

Experimentally, the `root_pcp()` approach to PCP modeling has best been able to handle those datasets that are governed by well-defined underlying patterns, characterized by quickly decaying singular values. This is typical of imaging and video data, but uncommon for EH data. For observed data with a complex low rank structure (slowly decaying singular values), like EH data, `rrmc()` may offer a better model estimate.

Three EH-specific extensions are currently supported by `root_pcp()`:

1. The model can handle missing values in the input data matrix `D`;
2. The model can also handle measurements that fall below the limit of detection (LOD), if provided `LOD` information by the user; and
3. The model is also equipped with an optional non-negativity constraint on the low-rank `L` matrix, ensuring that all output values in `L` are > 0.

Usage

```
root_pcp(
  D,
  lambda = NULL,
  mu = NULL,
  LOD = -Inf,
  non_negative = TRUE,
  max_iter = 10000,
  verbose = FALSE
)
```

Arguments

D	The input data matrix (can contain NA values). Note that PCP will converge much more quickly when D has been standardized in some way (e.g. scaling columns by their standard deviations, or column-wise min-max normalization).
lambda, mu	(Optional) A pair of doubles each in the range [0, Inf) regularizing S and L. lambda controls the sparsity of the output S matrix; larger values penalize non-zero entries in S more stringently, driving the recovery of sparser S matrices. mu adjusts the model's sensitivity to noise; larger values will penalize errors between the predicted model and the observed data more severely. It is highly recommended the user tunes both of these parameters using grid_search_cv() for each unique data matrix D. By default, both lambda and mu are NULL, in which case the theoretically optimal values are used, calculated according to get_pcp_defaults().
LOD	(Optional) The limit of detection (LOD) data. Entries in D that satisfy D >= LOD are understood to be above the LOD, otherwise those entries are treated as below the LOD. LOD can be either: • A double, implying a universal LOD common across all measurements in D; • A vector of length ncol(D), signifying a column-specific LOD, where each entry in the LOD vector corresponds to the LOD for each column in D; or • A matrix of dimension dim(D), indicating an observation-specific LOD, where each entry in the LOD matrix corresponds to the LOD for each entry in D. By default, LOD = -Inf, indicating there are no known LODs for PCP to leverage.
non_negative	(Optional) A logical indicating whether or not the non-negativity constraint should be used to constrain the output L matrix to have all entries ≥ 0. By default, non_negative = TRUE.
max_iter	(Optional) An integer specifying the maximum number of iterations to allow PCP before giving up on meeting PCP's convergence criteria. By default, max_iter = 10000, suitable for most problems.
verbose	(Optional) A logical indicating whether or not to print information in real time over the course of PCP's optimization. By default, verbose = FALSE.

Value

A list containing:

- `L`: The rank-`r` low-rank matrix encoding the `r`-many latent patterns governing the observed input data matrix `D`. `dim(L)` will be the same as `dim(D)`. To explicitly obtain the underlying patterns, `L` can be used as the input to any matrix factorization technique of choice, e.g. PCA, factor analysis, or non-negative matrix factorization.
- `S`: The sparse matrix containing the rare outlying or extreme observations in `D` that are not explained by the underlying patterns in the corresponding `L` matrix. `dim(S)` will be the same as `dim(D)`. Most entries in `S` are `0`, while non-zero entries identify the extreme outlying observations in `D`.
- `num_iter`: The number of iterations taken to reach convergence. If `num_iter == max_iter` then `root_pcp()` did not converge.
- `objective`: A vector containing the values of `root_pcp()`'s objective function over the course of optimization.
- `converged`: A boolean indicating whether the convergence criteria were met before `max_iter` was reached.

The objective function

`root_pcp()` optimizes the following objective function:

$$\min_{L,S} ||L||_* + \lambda||S||_1 + \mu||L + S - D||_F$$

The first term is the nuclear norm of the `L` matrix, incentivizing `L` to be low-rank. The second term is the ℓ_1 norm of the `S` matrix, encouraging `S` to be sparse. The third term is the Frobenius norm applied to the model's noise, ensuring that the estimated low-rank and sparse models `L` and `S` together have high fidelity to the observed data `D`. The objective is not smooth nor differentiable, however it is convex and separable. As such, it is optimized using the Alternating Direction Method of Multipliers (ADMM) algorithm Boyd et al. (2011), Gao et al. (2020).

The `lambda` and `mu` parameters

- `lambda` controls the sparsity of `root_pcp()`'s output `S` matrix; larger values of `lambda` penalize non-zero entries in `S` more stringently, driving the recovery of sparser `S` matrices. Therefore, if you a priori expect few outlying events in your model, you might expect a grid search to recover relatively larger `lambda` values, and vice-versa.
- `mu` adjusts `root_pcp()`'s sensitivity to noise; larger values of `mu` penalize errors between the predicted model and the observed data (i.e. noise), more severely. Environmental data subject to higher noise levels therefore require a `root_pcp()` model equipped with smaller `mu` values (since higher noise means a greater discrepancy between the observed mixture and the true underlying low-rank and sparse model). In virtually noise-free settings (e.g. simulations), larger values of `mu` would be appropriate.

The default values of `lambda` and `mu` offer *theoretical* guarantees of optimal estimation performance, and stable recovery of `L` and `S`. By "stable", we mean `root_pcp()`'s reconstruction error is, in the worst case, proportional to the magnitude of the noise corrupting the observed data ($||Z||_F$), often outperforming this upper bound. Candès et al. (2011) obtained the guarantee for `lambda`, while Zhang et al. (2021) obtained the result for `mu`.

Environmental health specific extensions

We refer interested readers to Gibson et al. (2022) for the complete details regarding the EH-specific extensions.

Missing value functionality: PCP assumes that the same data generating mechanisms govern both the missing and the observed entries in `D`. Because PCP primarily seeks accurate estimation of *patterns* rather than individual *observations*, this assumption is reasonable, but in some edge cases may not always be justified. Missing values in `D` are therefore reconstructed in the recovered low-rank `L` matrix according to the underlying patterns in `L`. There are three corollaries to keep in mind regarding the quality of recovered missing observations:

1. Recovery of missing entries in `D` relies on accurate estimation of `L`;
2. The fewer observations there are in `D`, the harder it is to accurately reconstruct `L` (therefore estimation of *both* unobserved *and* observed measurements in `L` degrades); and
3. Greater proportions of missingness in `D` artifically drive up the sparsity of the estimated `S` matrix. This is because it is not possible to recover a sparse event in `S` when the corresponding entry in `D` is unobserved. By definition, sparse events in `S` cannot be explained by the consistent patterns in `L`. Practically, if 20% of the entries in `D` are missing, then at least 20% of the entries in `S` will be 0.

Handling measurements below the limit of detection: When equipped with LOD information, PCP treats any estimations of values known to be below the LOD as equally valid if their approximations fall between 0 and the LOD. Over the course of optimization, observations below the LOD are pushed into this known range $[0, LOD]$ using penalties from above and below: should a $< LOD$ estimate be < 0, it is stringently penalized, since measured observations cannot be negative. On the other hand, if a $< LOD$ estimate is $>$ the LOD, it is also heavily penalized: less so than when < 0, but more so than observations known to be above the LOD, because we have prior information that these observations must be below LOD. Observations known to be above the LOD are penalized as usual, using the Frobenius norm in the above objective function.

Gibson et al. (2022) demonstrates that in experimental settings with up to 50% of the data corrupted below the LOD, PCP with the LOD extension boasts superior accuracy of recovered `L` models compared to PCA coupled with $LOD/\sqrt{2}$ imputation. PCP even outperforms PCA in low-noise scenarios with as much as 75% of the data corrupted below the LOD. The few situations in which PCA bettered PCP were those pathological cases in which `D` was characterized by extreme noise and huge proportions (i.e., 75%) of observations falling below the LOD.

The non-negativity constraint on `L`: To enhance interpretability of PCP-rendered solutions, there is an optional non-negativity constraint that can be imposed on the `L` matrix to ensure all estimated values within it are ≥ 0. This prevents researchers from having to deal with negative observation values and questions surrounding their meaning and utility. Non-negative `L` models also allow for seamless use of methods such as non-negative matrix factorization to extract non-negative patterns. The non-negativity constraint is incorporated in the ADMM splitting technique via the introduction of an additional optimization variable and corresponding constraint.

References

Zhang, Junhui, Jingkai Yan, and John Wright. "Square root principal component pursuit: tuning-free noisy robust matrix recovery." Advances in Neural Information Processing Systems 34 (2021): 29464-29475. [available here]

Gibson, Elizabeth A., Junhui Zhang, Jingkai Yan, Lawrence Chillrud, Jaime Benavides, Yanelli Nunez, Julie B. Herbstman, Jeff Goldsmith, John Wright, and Marianthi-Anna Kioumourtzoglou. "Principal component pursuit for pattern identification in environmental mixtures." Environmental Health Perspectives 130, no. 11 (2022): 117008.

Boyd, Stephen, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein. "Distributed optimization and statistical learning via the alternating direction method of multipliers." Foundations and Trends in Machine learning 3, no. 1 (2011): 1-122.

Gao, Wenbo, Donald Goldfarb, and Frank E. Curtis. "ADMM for multiaffine constrained optimization." Optimization Methods and Software 35, no. 2 (2020): 257-303.

Candès, Emmanuel J., Xiaodong Li, Yi Ma, and John Wright. "Robust principal component analysis?." Journal of the ACM (JACM) 58, no. 3 (2011): 1-37.

See Also

`rrmc()`

Examples

```
#### -------Simple simulated PCP problem-------####
# First we will simulate a simple dataset with the sim_data() function.
# The dataset will be a 100x10 matrix comprised of:
# 1. A rank-2 component as the ground truth L matrix;
# 2. A ground truth sparse component S w/outliers along the diagonal; and
# 3. A dense Gaussian noise component
data <- sim_data(r = 2, sigma = 0.1)
# Best practice is to conduct a grid search with grid_search_cv() function,
# but we skip that here for brevity.
pcp_model <- root_pcp(data$D, lambda = 0.225, mu = 3.04)
data.frame(
  "Estimated_L_rank" = matrix_rank(pcp_model$L, 5e-2),
  "Observed_relative_error" = norm(data$L - data$D, "F") / norm(data$L, "F"),
  "PCA_error" = norm(data$L - proj_rank_r(data$D, r = 2), "F") / norm(data$L, "F"),
  "PCP_L_error" = norm(data$L - pcp_model$L, "F") / norm(data$L, "F"),
  "PCP_S_error" = norm(data$S - pcp_model$S, "F") / norm(data$S, "F")
)
```

---

rrmc *Rank-based robust matrix completion (non-convex PCP)*

Description

`rrmc()` implements the non-convex PCP algorithm "Rank-based robust matrix completion" as described in Cherapanamjeri et al. (2017) (see Algorithm 3), outfitted with environmental health (EH)-specific extensions as described in Gibson et al. (2022).

Given an observed data matrix `D`, maximum rank to search up to `r`, and regularization parameter `eta`, `rrmc()` seeks to find the best low-rank and sparse estimates `L` and `S` using an incremental rank-based strategy. The `L` matrix encodes latent patterns that govern the observed data. The `S` matrix captures any extreme events in the data unexplained by the underlying patterns in `L`.

`rrmc()`'s incremental rank-based strategy first estimates a rank-1 model $(L^{(1)}, S^{(1)})$, before using the rank-1 model as the initialization point to then construct a rank-2 model $(L^{(2)}, S^{(2)})$, and so on, until the desired rank-`r` model $(L^{(r)}, S^{(r)})$ is recovered. All models from ranks 1 through `r` are returned by `rrmc()` in this way.

Experimentally, the `rrmc()` approach to PCP has best been able to handle those datasets that are governed by complex underlying patterns characterized by slowly decaying singular values, such as EH data. For observed data with a well-defined low rank structure (rapidly decaying singular values), `root_pcp()` may offer a better model estimate.

Two EH-specific extensions are currently supported by `rrmc()`:

1. The model can handle missing values in the input data matrix `D`; and
2. The model can also handle measurements that fall below the limit of detection (LOD), if provided `LOD` information by the user.

Support for a non-negativity constraint on `rrmc()`'s output will be added in a future release of `pcpr`.

Usage

```
rrmc(D, r, eta = NULL, LOD = -Inf)
```

Arguments

`D`	The input data matrix (can contain `NA` values). Note that PCP will converge much more quickly when `D` has been standardized in some way (e.g. scaling columns by their standard deviations, or column-wise min-max normalization).
`r`	An integer `>= 1` specifying the maximum rank PCP model to return. All models from rank `1` through `r` will be returned.
`eta`	(Optional) A double in the range `[0, Inf)` defining the ratio between the model's sensitivity to sparse and dense noise. Larger values of `eta` will place a greater emphasis on penalizing the non-zero entries in `S` over penalizing dense noise `Z`, i.e. errors between the predicted and observed data `Z = L + S - D`. It is recommended to tune `eta` using `grid_search_cv()` for each unique data matrix `D`. By default, `eta = NULL`, in which case `eta` is retrieved using `get_pcp_defaults()`.
`LOD`	(Optional) The limit of detection (LOD) data. Entries in `D` that satisfy `D >= LOD` are understood to be above the LOD, otherwise those entries are treated as below the LOD. `LOD` can be either: • A double, implying a universal LOD common across all measurements in `D`; • A vector of length `ncol(D)`, signifying a column-specific LOD, where each entry in the `LOD` vector corresponds to the LOD for each column in `D`; or • A matrix of dimension `dim(D)`, indicating an observation-specific LOD, where each entry in the `LOD` matrix corresponds to the LOD for each entry in `D`. By default, `LOD = -Inf`, indicating there are no known LODs for PCP to leverage.

Value

A list containing:

- `L`: The rank-`r` low-rank matrix encoding the `r`-many latent patterns governing the observed input data matrix `D`. `dim(L)` will be the same as `dim(D)`. To explicitly obtain the underlying patterns, `L` can be used as the input to any matrix factorization technique of choice, e.g. PCA, factor analysis, or non-negative matrix factorization.
- `S`: The sparse matrix containing the rare outlying or extreme observations in `D` that are not explained by the underlying patterns in the corresponding `L` matrix. `dim(S)` will be the same as `dim(D)`. Most entries in `S` are `0`, while non-zero entries identify the extreme outlying observations in `D`.
- `L_list`: A list of the `r`-many `L` matrices recovered over the course of `rrmc()`'s iterative optimization procedure. The first element in `L_list` corresponds to the rank-`1` `L` matrix, the second to the rank-`2` `L` matrix, and so on.
- `S_list`: A list of the `r`-many corresponding `S` matrices recovered over the course of `rrmc()`'s iterative optimization procedure. The first element in `S_list` corresponds to the rank-`1` solution's `S` matrix, the second to the rank-`2` solution's `S` matrix, and so on.
- `objective`: A vector containing the values of `rrmc()`'s objective function over the course of optimization.

The objective function

`rrmc()` implicitly optimizes the following objective function:

$$\min_{L,S} I_{rank(L)\leq r} + \eta||S||_0 + ||L + S - D||_F^2$$

The first term is the indicator function checking that the `L` matrix is strictly rank `r` or less, implemented using a rank `r` projection operator `proj_rank_r()`. The second term is the ℓ_0 norm applied to the `S` matrix to encourage sparsity, and is implemented with the help of an adaptive hard-thresholding operator `hard_threshold()`. The third term is the squared Frobenius norm applied to the model's noise.

The `eta` parameter

The `eta` parameter scales the sparse penalty applied to `rrmc()`'s output sparse `S` matrix. Larger values of `eta` penalize non-zero entries in `S` more stringently, driving the recovery of sparser `S` matrices.

Because there are no other parameters scaling the other terms in `rrmc()`'s objective function, `eta` can intuitively be thought of as the dial that balances the model's sensitivity to extreme events (placed in `S`) and its sensitivity to noise `Z` (captured by the last term in the objective, which measures the discrepancy between the between the predicted model and the observed data). Larger values of `eta` will place a greater emphasis on penalizing the non-zero entries in `S` over penalizing the errors between the predicted and observed data `Z = L + S - D`.

Environmental health specific extensions

We refer interested readers to Gibson et al. (2022) for the complete details regarding the EH-specific extensions.

Missing value functionality: PCP assumes that the same data generating mechanisms govern both the missing and the observed entries in D. Because PCP primarily seeks accurate estimation of *patterns* rather than individual *observations*, this assumption is reasonable, but in some edge cases may not always be justified. Missing values in D are therefore reconstructed in the recovered low-rank L matrix according to the underlying patterns in L. There are three corollaries to keep in mind regarding the quality of recovered missing observations:

1. Recovery of missing entries in D relies on accurate estimation of L;
2. The fewer observations there are in D, the harder it is to accurately reconstruct L (therefore estimation of *both* unobserved *and* observed measurements in L degrades); and
3. Greater proportions of missingness in D artifically drive up the sparsity of the estimated S matrix. This is because it is not possible to recover a sparse event in S when the corresponding entry in D is unobserved. By definition, sparse events in S cannot be explained by the consistent patterns in L. Practically, if 20% of the entries in D are missing, then at least 20% of the entries in S will be 0.

Handling measurements below the limit of detection: When equipped with LOD information, PCP treats any estimations of values known to be below the LOD as equally valid if their approximations fall between 0 and the LOD. Over the course of optimization, observations below the LOD are pushed into this known range $[0, LOD]$ using penalties from above and below: should a $< LOD$ estimate be < 0, it is stringently penalized, since measured observations cannot be negative. On the other hand, if a $< LOD$ estimate is $>$ the LOD, it is also heavily penalized: less so than when < 0, but more so than observations known to be above the LOD, because we have prior information that these observations must be below LOD. Observations known to be above the LOD are penalized as usual, using the Frobenius norm in the above objective function.

Gibson et al. (2022) demonstrates that in experimental settings with up to 50% of the data corrupted below the LOD, PCP with the LOD extension boasts superior accuracy of recovered L models compared to PCA coupled with $LOD/\sqrt{2}$ imputation. PCP even outperforms PCA in low-noise scenarios with as much as 75% of the data corrupted below the LOD. The few situations in which PCA bettered PCP were those pathological cases in which D was characterized by extreme noise and huge proportions (i.e., 75%) of observations falling below the LOD.

References

Cherapanamjeri, Yeshwanth, Kartik Gupta, and Prateek Jain. "Nearly optimal robust matrix completion." International Conference on Machine Learning. PMLR, 2017. [available here]

Gibson, Elizabeth A., Junhui Zhang, Jingkai Yan, Lawrence Chillrud, Jaime Benavides, Yanelli Nunez, Julie B. Herbstman, Jeff Goldsmith, John Wright, and Marianthi-Anna Kioumourtzoglou. "Principal component pursuit for pattern identification in environmental mixtures." Environmental Health Perspectives 130, no. 11 (2022): 117008.

See Also

root_pcp()

Examples

```
#### -------Simple simulated PCP problem-------####
# First we will simulate a simple dataset with the sim_data() function.
```

```
# The dataset will be a 100x10 matrix comprised of:
# 1. A rank-3 component as the ground truth L matrix;
# 2. A ground truth sparse component S w/outliers along the diagonal; and
# 3. A dense Gaussian noise component
data <- sim_data()
# Best practice is to conduct a grid search with grid_search_cv() function,
# but we skip that here for brevity.
pcp_model <- rrmc(data$D, r = 3, eta = 0.35)
data.frame(
  "Observed_relative_error" = norm(data$L - data$D, "F") / norm(data$L, "F"),
  "PCA_error" = norm(data$L - proj_rank_r(data$D, r = 3), "F") / norm(data$L, "F"),
  "PCP_L_error" = norm(data$L - pcp_model$L, "F") / norm(data$L, "F"),
  "PCP_S_error" = norm(data$S - pcp_model$S, "F") / norm(data$S, "F")
)
```

`sim_data` *Simulate simple mixtures data*

Description

`sim_data()` generates a simulated dataset `D = L + S + Z` for experimentation with Principal Component Pursuit (PCP) algorithms.

Usage

```
sim_data(
  n = 100,
  p = 10,
  r = 3,
  sparse_nonzero_idxs = NULL,
  sigma = 0.05,
  seed = 42
)
```

Arguments

`n, p`	(Optional) A pair of integers specifying the simulated dataset's number of `n` observations (rows) and `p` variables (columns). By default, `n = 100`, and `p = 10`.
`r`	(Optional) An integer specifying the rank of the simulated dataset's low-rank component. Intuitively, the number of latent patterns governing the simulated dataset. Must be that `r <= min(n, p)`. By default, `r = 3`.
`sparse_nonzero_idxs`	(Optional) An integer vector with `length(sparse_nonzero_idxs) <= n * p` specifying the indices of the non-zero elements in the sparse component. By default, `sparse_nonzero_idxs = NULL`, in which case it is defined to be the vector `seq(1, n * p, n + 1)` (placing sparse noise along the diagonal of the simulated dataset).

sigma	(Optional) A double specifying the standard deviation of the dense (Gaussian) noise component Z. By default, sigma = 0.05.
seed	(Optional) An integer specifying the seed for random number generation. By default, seed = 42.

Details

The data is simulated as follows:

```
L <- matrix(runif(n * r), n, r) %*% matrix(runif(r * p), r, p)

S <- matrix(0, n, p)

S[sparse_nonzero_idxs] <- 1

Z <- matrix(rnorm(n * p, sd = sigma), n, p)

D <- L + S + Z
```

Value

A list containing:

- D: The observed data matrix, where D = L + S + Z.
- L: The ground truth rank-r low-rank matrix.
- S: The ground truth sparse matrix.
- S: The ground truth dense (Gaussian) noise matrix.

See Also

sim_na(), sim_lod(), impute_matrix()

Examples

```
# rank 3 example
data <- sim_data()
matrix_rank(data$D)
matrix_rank(data$L)
# rank 7 example
data <- sim_data(n = 1000, p = 25, r = 7)
matrix_rank(data$D)
matrix_rank(data$L)
```

`sim_lod` *Simulate limit of detection data*

Description

`sim_lod()` simulates putting the columns of a given matrix `D` under a limit of detection (LOD) by calculating the given quantile `q` of each column and corrupting all values < the quantile to `NA`, returning the newly corrupted matrix, the binary corruption mask, and a vector of column LODs.

Usage

```
sim_lod(D, q)
```

Arguments

`D`	The input data matrix.
`q`	A double in the range `[0, 1]` specifying the quantile to use in creating the column-wise LODs. Passed as the `probs` argument to the `quantile()` function.

Value

A list containing:

- `D_tilde`: The original matrix `D` corrupted with < LOD `NA` values.
- `tilde_mask`: A binary matrix of `dim(D)` specifying the locations of corrupted entries (`1`) and uncorrupted entries (`0`).
- `lod`: A vector with `length(lod) == ncol(D)` providing the simulated LOD values corresponding to each column in the `D_tilde`.

See Also

`sim_na()`, `impute_matrix()`, `sim_data()`

Examples

```
D <- sim_data(5, 5, sigma = 0.8)$D
D
sim_lod(D, q = 0.2)
```

`sim_na` *Simulate random missingness in a given matrix*

Description

`sim_na()` corrupts a given data matrix `D` such that a random `perc` percent of its entries are set to be missing (set to `NA`). Used by `grid_search_cv()` in constructing test matrices for PCP models. Can be used for experimentation with PCP models.

Note: only *observed* values can be corrupted as `NA`. This means if a matrix `D` already has e.g. 20% of its values missing, then `sim_na(D, perc = 0.2)` would result in a matrix with 40% of its values as missing.

Should e.g. `perc = 0.6` be passed as input when `D` only has e.g. 10% of its entries left as observed, then all remaining corruptable entries will be set to `NA`.

Usage

```
sim_na(D, perc, seed = 42)
```

Arguments

`D`	The input data matrix.
`perc`	A double in the range `[0, 1]` specifying the percentage of entries in `D` to corrupt as missing (`NA`).
`seed`	(Optional) An integer specifying the seed for the random selection of entries in `D` to corrupt as missing (`NA`). By default, `seed = 42`.

Value

A list containing:

- `D_tilde`: The original matrix `D` with a random `perc` percent of its entries set to `NA`.
- `tilde_mask`: A binary matrix of `dim(D)` specifying the locations of corrupted entries (`1`) and uncorrupted entries (`0`).

See Also

`grid_search_cv()`, `sim_lod()`, `impute_matrix()`, `sim_data()`

Examples

```
# Simple example corrupting 20% of a 5x5 matrix
D <- matrix(1:25, 5, 5)
corrupted_data <- sim_na(D, perc = 0.2)
corrupted_data$D_tilde
sum(is.na(corrupted_data$D_tilde)) / prod(dim(corrupted_data$D_tilde))
# Now corrupting another 20% ontop of the original 20%
double_corrupted <- sim_na(corrupted_data$D_tilde, perc = 0.2)
```

```
double_corrupted$D_tilde
sum(is.na(double_corrupted$D_tilde)) / prod(dim(double_corrupted$D_tilde))
# Corrupting the remaining entries by passing in a large value for perc
all_corrupted <- sim_na(double_corrupted$D_tilde, perc = 1)
all_corrupted$D_tilde
```

`sing` *Compute singular values of given matrix*

Description

`sing()` calculates the singular values of a given data matrix `D`. This is done with a call to `svd()`, and is included in `pcpr` to enable the quick characterization of a data matrix's raw low-rank structure, to help decide whether `rrmc()` or `root_pcp()` is the more appropriate PCP algorithm to employ in conjunction with `D`.

Experimentally, the `rrmc()` approach to PCP has best been able to handle those datasets that are governed by complex underlying patterns characterized by slowly decaying singular values, such as EH data. For observed data with a well-defined low rank structure (rapidly decaying singular values), `root_pcp()` may offer a better model estimate.

Usage

```
sing(D)
```

Arguments

`D`	The input data matrix (cannot have `NA` values).

Value

A numeric vector containing the singular values of `D`.

References

"Singular value decomposition" Wikipedia article.

See Also

`matrix_rank()`

Examples

```
data <- sim_data()
sing(data$D)
```

sparsity *Estimate sparsity of given matrix*

Description

`sparsity()` estimates the percentage of entries in a given data matrix `D` whose values are "practically zero". If the absolute value of an entry is below a given threshold parameter `thresh`, then that value is determined to be "practically zero", increasing the estimated sparsity of `D`. *Note that* `NA` *values are imputed as 0 before the sparsity calculation is made.*

Usage

```
sparsity(D, thresh = 1e-04)
```

Arguments

`D`	The input data matrix.
`thresh`	(Optional) A numeric threshold `>= 0` used to determine if an entry in `D` is "practically zero". If the absolute value of an entry is below `thresh`, then it is judged to be "practically zero". By default, `thresh = 1e-04`.

Value

The sparsity of `D`, measured as the percentage of entries in `D` that are "practically zero".

See Also

`matrix_rank()`

Examples

```
sparsity(matrix(rep(c(1, 0), 8), 4, 4))
sparsity(matrix(0:8, 3, 3))
sparsity(matrix(0, 3, 3))
```

Index