

Package: RcppML (via r-universe)

October 19, 2026

Type Package

Title Fast Non-Negative Matrix Factorization and Divisive Clustering

Version 1.0.0

Date 2026-03-10

Description High-performance non-negative matrix factorization (NMF), singular value decomposition (SVD), and divisive clustering for large sparse and dense matrices. Implements alternating least squares with coordinate descent and Cholesky NNLS solvers, diagonal scaling for interpretable factors, cross-validation for automatic rank selection, multiple distribution-based losses (Gaussian, Poisson, Generalized Poisson, Negative Binomial, Gamma, Inverse Gaussian, Tweedie) via iteratively reweighted least squares, regularization (L1, L2, L21, angular, graph Laplacian), and optional GPU acceleration via CUDA. Includes divisive clustering via recursive rank-2 factorization, consensus clustering, and the StreamPress compressed sparse matrix format. Methods are described in DeBruine, Melcher, and Triche (2021) [<doi:10.1101/2021.09.01.458620>](https://doi.org/10.1101/2021.09.01.458620).

Depends R (>= 4.1.0), Matrix

License GPL (>= 3)

Imports Rcpp, methods, stats, utils

LinkingTo Rcpp, RcppEigen

RoxygenNote 7.3.3

Suggests ggplot2, hdf5r, jsonlite, knitr, pheatmap, randomForest, RColorBrewer, rmarkdown, patchwork, plotly, scales, Seurat, SeuratData, SeuratObject, testthat (>= 3.0.0), uwot, viridis

VignetteBuilder knitr

Config/testthat/edition 3

LazyData true

LazyDataCompression xz

URL <https://github.com/zdebruine/RcppML>

BugReports <https://github.com/zdebruine/RcppML/issues>

SystemRequirements CUDA Toolkit >= 11.0 (optional, for GPU acceleration)

Encoding UTF-8

Config/pak/sysreqs nvidia-cuda-dev

Repository <https://zdebruine.r-universe.dev>

Date/Publication 2026-10-19 17:24:49 UTC

RemoteUrl <https://github.com/zdebruine/rcppml>

RemoteRef HEAD

RemoteSha 49df8da602fac19aa3932623ef1932723a978a01

Contents

RcppML-package	4
\$.factor_net_result	6
align	6
aml	7
as.data.frame.nmf_assessment	8
as.data.frame.training_logger	9
assess	9
auto_nmf_distribution	11
bipartiteMatch	13
bipartition	14
biplot,nmf-method	16
classify_embedding	17
classify_logistic	18
classify_rf	19
compare_nmf	20
compute_target	21
consensus_nmf	22
cosine	24
cross_validate_graph	25
dclust	27
diagnose_dispersion	29
diagnose_zero_inflation	30
digits	31
evaluate	32
export_log	34
factor_add	34
factor_concat	35
factor_condition	36
factor_config	36
factor_input	38

factor_net	39
factor_shared	40
fit	41
golub	42
gpu_available	43
gpu_info	44
gpu_sparse_matrix-methods	44
hawaiibirds	45
movielens	47
nmf	48
nmf-class	53
nmf_layer	54
nnls	55
olivetti	58
pbmc3k	59
pca	61
plot.consensus_nmf	61
plot.dclust	63
plot.nmf	64
plot.nmfCrossValidate	65
plot.training_logger	66
predict.factor_net_result	66
print.factor_net	67
print.factor_net_cv	68
print.factor_net_result	68
print.fn_classifier_eval	69
print.fn_factor_config	69
print.fn_global_config	70
print.fn_node	71
print.nmf_assessment	71
print.training_logger	72
refine	72
score_test_distribution	74
simulateNMF	76
simulateSwimmer	77
sparsity	78
st_add_transpose	79
st_chunk_ranges	80
st_filter_cols	81
st_filter_rows	81
st_free_gpu	82
st_info	83
st_map_chunks	84
st_obs_indices	85
st_read	85
st_read_dense	86
st_read_gpu	87
st_read_obs	88

st_read_var	89
st_slice	90
st_slice_cols	90
st_slice_rows	91
st_write	92
st_write_dense	93
st_write_list	95
streampress	96
subset,nmf-method	96
summary,nmf-method	100
summary.consensus_nmf	101
summary.factor_net_result	102
summary.fn_classifier_eval	102
svd	103
svd-class	106
svd_layer	108
training_logger	109
W	110

Index	112
--------------	------------

RcppML-package	<i>RcppML: Fast Non-Negative Matrix Factorization and Divisive Clustering</i>
----------------	---

Description

High-performance non-negative matrix factorization (NMF), singular value decomposition (SVD/PCA), and divisive clustering for large sparse and dense matrices, powered by Rcpp and Eigen.

NMF (Non-negative Matrix Factorization)

`nmf` Fit NMF model (sparse or dense input, optional cross-validation)
`evaluate` Evaluate reconstruction loss of an NMF model
`align` Align factors across NMF models
`predict,nmf-method` Project new data onto a fitted NMF model
`consensus_nmf` Consensus clustering from multiple NMF runs
`simulateNMF` Simulate data from a known NMF model
`auto_nmf_distribution` Select distribution based on data characteristics

SVD / PCA

`svd` Truncated SVD via deflation
`pca` PCA (centered SVD)
`reconstruct` Reconstruct matrix from SVD/PCA model
`variance_explained` Proportion of variance per factor

NNLS (Non-negative Least Squares)

`nnls` Solve non-negative least squares problems

Clustering

`dclust` Divisive clustering via recursive rank-2 NMF

`bipartition` Split samples into two groups via rank-2 NMF

`bipartiteMatch` Match two sets of cluster labels

Factor Networks (multi-layer / multi-modal)

`factor_net` Compile a factorization network

`fit` Fit a compiled factor network

`factor_input`, `nmf_layer`, `svd_layer` Node constructors

`factor_shared`, `factor_concat`, `factor_add` Merge operations

`factor_config`, `W`, `H` Configuration

`cross_validate_graph` Cross-validate a factor network

StreamPress I/O

`st_write`, `st_read` Read/write .spz files

`st_info` Inspect .spz file metadata

`st_read_obs`, `st_read_var` Read embedded metadata tables

`st_read_gpu`, `st_free_gpu` GPU-direct .spz reading

GPU

`gpu_available` Check GPU availability

`gpu_info` Get GPU device details

Utilities

`cosine` Cosine similarity

`sparsity` Matrix sparsity fraction

Author(s)

Maintainer: Zachary DeBruine <zacharydebruine@gmail.com> ([ORCID](#))

See Also

Useful links:

- <https://github.com/zdebruine/RcppML>
- Report bugs at <https://github.com/zdebruine/RcppML/issues>

<code>\$.factor_net_result</code>	<i>Access layer results by name</i>
-----------------------------------	-------------------------------------

Description

Access layer results by name

Usage

```
## S3 method for class 'factor_net_result'  
x$name
```

Arguments

x	A factor_net_result object.
name	Layer name (e.g. "L1").

Value

The layer result list, or the named field if not a layer name.

See Also

[factor_net](#), [fit](#)

<code>align</code>	<i>Align two NMF models</i>
--------------------	-----------------------------

Description

Align two NMF models

Usage

```
align(object, ...)  
  
## S4 method for signature 'nmf'  
align(object, ref, method = "cosine", ...)
```

Arguments

object	nmf model to be aligned to ref
...	arguments passed to or from other methods
ref	reference nmf model to which object will be aligned
method	either cosine or cor

Details

For `nmf` models, factors in object are reordered to minimize the cost of bipartite matching (see `bipartiteMatch`) on a `cosine` or correlation distance matrix. The w matrix is used for matching, and must be equidimensional in object and ref.

Value

An `nmf` object with factors reordered to best match ref.

See Also

`bipartiteMatch`, `cosine`, `nmf`

Examples

```
data <- simulateNMF(50, 30, k = 3, seed = 1)
m1 <- nmf(data$A, 3, seed = 1, maxit = 50)
m2 <- nmf(data$A, 3, seed = 2, maxit = 50)
aligned <- align(m2, m1) # reorder m2 factors to match m1
```

aml

Acute Myelogenous Leukemia (AML) Dataset

Description

ATAC-seq chromatin accessibility data from 123 acute myelogenous leukemia (AML) patient samples and 5 healthy hematopoietic stem/progenitor cell (HSPC) samples. The dataset contains chromatin accessibility measurements across 824 genomic regions.

Usage

```
aml
```

Format

A matrix (dense) with 824 rows (genomic regions) and 135 columns (samples). Sample metadata is stored as an attribute. The data is stored as dense because it has very low sparsity (~0.2% zeros).

Access metadata via `attr(aml, "metadata_h")`, which contains:

samples Character vector of sample IDs

category Character vector indicating sample type (AML subtype or HSPC)

Details

This dataset contains ATAC-seq chromatin accessibility data for studying acute myelogenous leukemia. Samples represent different AML subtypes including:

- Common myeloid progenitors (CMP)
- Granulocyte-monocyte progenitors (GMP)
- Lymphoid-primed multi-potent progenitors (LMPP)
- Megakaryocyte-erythrocyte progenitors (MEP)
- Multi-potent progenitors (MPP)
- Healthy hematopoietic stem/progenitor cells (HSPC)

Source

Corces et al. (2016). "Lineage-specific and single-cell chromatin accessibility charts human hematopoiesis and leukemia evolution." *Nature Genetics* 48(10): 1193-1203.

Examples

```
# Load dataset
data(aml)

# Inspect structure
dim(aml)
table(attr(aml, "metadata_h")$category)

# Run NMF (transpose so samples are columns)
result <- nmf(t(aml), k = 6, maxit = 100)
```

```
as.data.frame.nmf_assessment
```

Convert assessment results to a one-row data frame

Description

Convert assessment results to a one-row data frame

Usage

```
## S3 method for class 'nmf_assessment'
as.data.frame(x, ...)
```

Arguments

x	An nmf_assessment object.
...	Ignored.

Value

A one-row data frame with all metrics as columns.

as.data.frame.training_logger
<i>Convert training log to data.frame</i>

Description

Converts the internal training log from an NMF run into a tidy data frame with one row per logged iteration, suitable for plotting convergence curves.

Usage

```
## S3 method for class 'training_logger'
as.data.frame(x, ...)
```

Arguments

- x A training_logger object (from nmf(..., log = TRUE)).
- ... Ignored.

Value

A data.frame with columns iteration, wall_sec, and optionally total_loss, per-layer loss columns, and norm-tracking columns.

See Also

```
nmf, plot.training_logger
```

assess	<i>Assess Embedding Quality</i>
--------	---------------------------------

Description

Unified evaluation of an NMF, SVD, or arbitrary embedding matrix against one or more label vectors. Computes clustering metrics (ARI, NMI, Silhouette), classification metrics (KNN/LR/RF accuracy, F1, AUC via cross-validation), and batch-mixing metrics (donor/batch Silhouette, kNN entropy).

Usage

```

assess(
  x,
  labels,
  batch = NULL,
  metrics = "all",
  n_folds = 5L,
  classifiers = c("knn", "lr", "rf"),
  k_nn = 15L,
  seed = 42L,
  min_class_size = 10L,
  sil_samples_per_class = 200L,
  batch_knn_k = 50L
)

```

Arguments

<code>x</code>	An object of class <code>nmf</code> , <code>svd</code> , or a numeric matrix (samples x features). For <code>nmf</code> objects, uses <code>t(diag(d) %*% h)</code> ; for <code>svd</code> objects, uses <code>u %*% diag(d)</code> .
<code>labels</code>	A factor, character, or integer vector of primary labels (e.g. cell type). Required for clustering and classification metrics.
<code>batch</code>	Optional factor/character/integer vector of batch labels (e.g. <code>donor_id</code> , <code>tissue</code>). When provided, batch-mixing metrics are computed.
<code>metrics</code>	Character vector specifying which metrics to compute. Options: <code>"ari"</code> , <code>"nmi"</code> , <code>"silhouette"</code> , <code>"classification"</code> , <code>"batch_mixing"</code> , or <code>"all"</code> (default).
<code>n_folds</code>	Number of cross-validation folds for classification (default 5).
<code>classifiers</code>	Character vector of classifiers: <code>"knn"</code> , <code>"lr"</code> , <code>"rf"</code> , or any combination (default all three).
<code>k_nn</code>	Number of neighbors for kNN classifier (default 15).
<code>seed</code>	Random seed for reproducibility (default 42).
<code>min_class_size</code>	Minimum number of samples per class. Classes smaller than this are dropped before evaluation (default 10).
<code>sil_samples_per_class</code>	Number of reference samples per class for approximate silhouette (default 200). Higher = more accurate, slower.
<code>batch_knn_k</code>	Number of neighbors for batch entropy (default 50).

Details

Uses GPU-accelerated kernels when available, otherwise falls back to CPU-based approximate algorithms that avoid $O(n^2)$ distance matrices:

- Silhouette: sampled approximation ($O(n * \text{samples_per_class} * C)$)
- kNN metrics: brute-force on GPU or sampled on CPU ($O(n * k * n_{\text{ref}})$)
- ARI/NMI: contingency table ($O(n)$, always fast)

Value

An S3 object of class `nmf_assessment` containing:

metrics Named list of all computed metric values

classification Data frame of per-classifier, per-fold results

params List recording all parameters used

Examples

```
# Assess an NMF model
library(Matrix)
A <- abs(rsparsematrix(200, 100, 0.2))
model <- nmf(A, 5, seed = 1)
labels <- factor(sample(letters[1:4], 100, replace = TRUE))
result <- assess(model, labels)
print(result)

# Select specific metrics
result <- assess(model, labels, metrics = c("nmi", "ari"))

# Include batch assessment
batch <- factor(sample(c("A", "B"), 100, replace = TRUE))
result <- assess(model, labels, batch = batch, metrics = "all")
```

`auto_nmf_distribution` *Auto-select NMF distribution*

Description

Fits NMF with multiple loss functions (distributions) and selects the best based on per-element AIC/BIC. Useful for determining whether count data is best modeled with Gaussian (MSE), Poisson/GP, or Negative Binomial loss.

Usage

```
auto_nmf_distribution(
  data,
  k,
  distributions = c("mse", "gp", "nb"),
  criterion = c("bic", "aic"),
  maxit = 50,
  seed = NULL,
  verbose = FALSE,
  ...
)
```

Arguments

<code>data</code>	Input matrix (dense or sparse dgCMatrix)
<code>k</code>	Factorization rank
<code>distributions</code>	Character vector of distributions to compare. Default: <code>c("mse", "gp", "nb")</code>
<code>criterion</code>	Selection criterion: "bic" (default) or "aic"
<code>maxit</code>	Maximum iterations per fit
<code>seed</code>	Random seed for reproducibility
<code>verbose</code>	Print progress and comparison table
<code>...</code>	Additional arguments passed to <code>nmf</code>

Details

For each distribution, NMF is fit and the final negative log-likelihood (NLL) is computed. For GP and NB, the C++ loss is already the total NLL. For MSE (Gaussian), the C++ loss is the sum of squared errors (SSE), which is converted to Gaussian NLL: $NLL = (N/2)(1 + \log(2\pi \cdot SSE/N))$.

The number of effective parameters is:

- mse: $k(m + n) + 1$ (factor params + noise variance)
- gp: $k(m + n) + m$ (factor params + dispersion per row)
- nb: $k(m + n) + m$ (factor params + size per row)

BIC = $2 \times NLL + df \times \log(N)$ where N is the number of observations (nonzeros for sparse, all entries for dense). AIC = $2 \times NLL + 2 \times df$.

Value

A list with:

loss Character string: name of the best distribution (loss function)

comparison Data frame with distribution, nll, df, aic, bic, selected

models Named list of fitted nmf objects

See Also

[score_test_distribution](#), [diagnose_zero_inflation](#), [nmf](#)

Examples

```
library(Matrix)
set.seed(42)
A <- abs(rsparsematrix(100, 50, 0.3))
result <- auto_nmf_distribution(A, k = 5)
print(result$comparison)
cat("Best distribution:", result$loss, "\n")
```

bipartiteMatch	<i>Bipartite graph matching</i>
----------------	---------------------------------

Description

Hungarian algorithm for matching samples in a bipartite graph from a distance ("cost") matrix

Usage

```
bipartiteMatch(x)
```

Arguments

`x` symmetric matrix giving the cost of every possible pairing

Details

This implementation was adapted from RcppHungarian, an Rcpp wrapper for the original C++ implementation by Cong Ma (2016).

Value

List with elements `cost` (total matching cost) and `assignment` (0-indexed column assignment for each row).

See Also

[align](#), [consensus_nmf](#)

Examples

```
cost <- matrix(c(1, 0.5, 0.2,
                 0.5, 1, 0.3,
                 0.2, 0.3, 1), nrow = 3, byrow = TRUE)
result <- bipartiteMatch(cost)
result$cost
result$assignment
```

bipartition	<i>Bipartition a sample set</i>
-------------	---------------------------------

Description

Spectral bipartitioning by rank-2 matrix factorization

Usage

```
bipartition(
  data,
  tol = 1e-05,
  nonneg = TRUE,
  threads = 0,
  verbose = FALSE,
  ...
)
```

Arguments

data	dense or sparse matrix of features in rows and samples in columns. Prefer matrix or Matrix::dgCMatrix, respectively. Also accepts a file path (character string) which will be auto-loaded based on extension.
tol	tolerance of the fit (default 1e-4)
nonneg	enforce non-negativity of the rank-2 factorization used for bipartitioning
threads	number of threads for OpenMP parallelization (default 0 = all available)
verbose	print progress information (default FALSE)
...	additional arguments (see Advanced Parameters section)

Details

Spectral bipartitioning is a popular subroutine in divisive clustering. The sign of the difference between sample loadings in factors of a rank-2 matrix factorization gives a bipartition that is nearly identical to an SVD.

Rank-2 matrix factorization by alternating least squares is faster than rank-2-truncated SVD (i.e. *irlba*).

This function is a specialization of rank-2 [nmf](#) with support for factorization of only a subset of samples, and with additional calculations on the factorization model relevant to bipartitioning. See [nmf](#) for details regarding rank-2 factorization.

Value

A list giving the bipartition and useful statistics:

- `v` : vector giving difference between sample loadings between factors in a rank-2 factorization

- `dist` : relative cosine distance of samples within a cluster to centroids of assigned vs. not-assigned cluster
- `size1` : number of samples in first cluster (positive loadings in 'v')
- `size2` : number of samples in second cluster (negative loadings in 'v')
- `samples1`: indices of samples in first cluster
- `samples2`: indices of samples in second cluster
- `center1` : mean feature loadings across samples in first cluster
- `center2` : mean feature loadings across samples in second cluster

Advanced Parameters

Several parameters may be specified in the `...` argument:

- `diag = TRUE`: scale factors in w and h to sum to 1 by introducing a diagonal, d . This should generally never be set to `FALSE`. Diagonalization enables symmetry of models in factorization of symmetric matrices, convex L1 regularization, and consistent factor scalings.
- `samples = 1:ncol(A)`: samples to include in bipartition, numbered from 1 to `ncol(A)`. Default is all samples.
- `calc_dist = TRUE`: calculate the relative cosine distance of samples within a cluster to either cluster centroid. If `TRUE`, centers for clusters will also be calculated.
- `seed = NULL`: random seed for model initialization, generally not needed for rank-2 factorizations because robust solutions are recovered when `diag = TRUE`
- `maxit = 100`: maximum number of alternating updates of w and h . Generally, rank-2 factorizations converge quickly and this should not need to be adjusted.

Note

`bipartition()` uses scalar nonneg (not length-2) because rank-2 factorizations apply the same constraint to both factors. The default `tol = 1e-5` (inherited from `nmf()`'s internal rank-2 path) is tighter than `nmf()`'s global `1e-4` because single rank-2 subproblems converge faster.

Author(s)

Zach DeBruine

References

Kuang, D, Park, H. (2013). "Fast rank-2 nonnegative matrix factorization for hierarchical document clustering." Proc. 19th ACM SIGKDD intl. conf. on Knowledge discovery and data mining.

See Also

[nmf](#), [dclust](#)

Examples

```
library(Matrix)
data(iris)
A <- as(as.matrix(iris[,1:4]), "dgCMatrix")
bipartition(A, calc_dist = TRUE)
```

biplot,nmf-method	<i>Biplot for NMF factors</i>
-------------------	-------------------------------

Description

Produces a biplot from the output of [nmf](#)

Usage

```
## S4 method for signature 'nmf'
biplot(x, factors = c(1, 2), matrix = "w", group_by = NULL, ...)
```

Arguments

x	an object of class "nmf"
factors	length 2 vector specifying factors to plot.
matrix	either w or h
group_by	a discrete factor giving groupings for samples or features. Must be of the same length as number of samples in object\$h or number of features in object\$w.
...	for consistency with biplot generic

Value

ggplot2 object

See Also

[nmf](#)

Examples

```
library(Matrix)
A <- rsparsematrix(100, 50, 0.1)
model <- nmf(A, k = 3, seed = 42, tol = 1e-2, maxit = 10)
if (requireNamespace("ggplot2", quietly = TRUE)) {
  biplot(model, factors = c(1, 2))
}
```

classify_embedding	<i>Evaluate classification performance of factor embeddings</i>
--------------------	---

Description

Trains a k-nearest-neighbor classifier on factor embeddings and evaluates on held-out test samples. Returns a comprehensive metrics object.

Usage

```
classify_embedding(
  embedding,
  labels,
  test_fraction = 0.2,
  test_idx = NULL,
  k = 5L,
  seed = NULL,
  distance = c("euclidean", "cosine")
)
```

Arguments

embedding	Numeric matrix where rows are samples and columns are features (e.g., <code>t(result\$H)</code> for sample embeddings).
labels	Integer or factor vector of class labels. Length must equal <code>nrow(embedding)</code> .
test_fraction	Fraction of samples held out for testing (default 0.2).
test_idx	Optional integer vector of test indices. If provided, <code>test_fraction</code> is ignored.
k	Number of nearest neighbors (default 5).
seed	Random seed for train/test split reproducibility.
distance	Distance metric: "euclidean" (default) or "cosine".

Value

An `fn_classifier_eval` object with fields:

- accuracy** Overall test accuracy
- per_class** Data frame with per-class precision, recall, F1, support
- macro_precision** Macro-averaged precision
- macro_recall** Macro-averaged recall
- macro_f1** Macro-averaged F1
- weighted_f1** Support-weighted F1
- auc** Macro-averaged one-vs-rest AUC (from neighbor vote fractions)
- confusion** Confusion matrix (rows = true, cols = predicted)

predictions Test set predictions
test_labels True test labels
train_idx Training sample indices
test_idx Test sample indices
k Number of neighbors used

See Also

[classify_logistic](#), [classify_rf](#)

Examples

```
data(digits)
model <- nmf(digits, 10, maxit = 20, seed = 1, verbose = FALSE)
labels <- attr(digits, "target")
eval <- classify_embedding(model$w, labels, test_fraction = 0.2, k = 5, seed = 42)
print(eval)
```

classify_logistic	<i>Logistic regression classifier for factor embeddings</i>
-------------------	---

Description

Trains a multinomial logistic regression on factor embeddings using `stats::glm` (one-vs-rest for > 2 classes) and evaluates on test samples with the same comprehensive metrics as `classify_embedding`.

Usage

```
classify_logistic(
  embedding,
  labels,
  test_fraction = 0.2,
  test_idx = NULL,
  seed = NULL
)
```

Arguments

embedding	Numeric matrix where rows are samples and columns are features (e.g., <code>t(result\$H)</code> for sample embeddings).
labels	Integer or factor vector of class labels. Length must equal <code>nrow(embedding)</code> .
test_fraction	Fraction of samples held out for testing (default 0.2).
test_idx	Optional integer vector of test indices. If provided, <code>test_fraction</code> is ignored.
seed	Random seed for train/test split reproducibility.

Value

An `fn_classifier_eval` object (same structure as KNN variant).

See Also

[classify_embedding](#), [classify_rf](#)

Examples

```
# Logistic regression on random embeddings
set.seed(42)
embed <- matrix(rnorm(200), nrow = 40, ncol = 5)
labels <- factor(rep(1:4, each = 10))
eval <- classify_logistic(embed, labels, seed = 1)
eval$accuracy
```

`classify_rf`

Random forest classifier for factor embeddings

Description

Trains a random forest on factor embeddings using the **randomForest** package (must be installed) and evaluates on test samples.

Usage

```
classify_rf(
  embedding,
  labels,
  test_fraction = 0.2,
  test_idx = NULL,
  ntree = 500L,
  seed = NULL
)
```

Arguments

<code>embedding</code>	Numeric matrix where rows are samples and columns are features (e.g., <code>t(result\$H)</code> for sample embeddings).
<code>labels</code>	Integer or factor vector of class labels. Length must equal <code>nrow(embedding)</code> .
<code>test_fraction</code>	Fraction of samples held out for testing (default 0.2).
<code>test_idx</code>	Optional integer vector of test indices. If provided, <code>test_fraction</code> is ignored.
<code>ntree</code>	Number of trees (default 500).
<code>seed</code>	Random seed for train/test split reproducibility.

Value

An `fn_classifier_eval` object (same structure as KNN variant).

See Also

[classify_embedding](#), [classify_logistic](#)

Examples

```
# Random forest on random embeddings (requires randomForest package)
if (requireNamespace("randomForest", quietly = TRUE)) {
  set.seed(42)
  embed <- matrix(rnorm(200), nrow = 40, ncol = 5)
  labels <- factor(rep(1:4, each = 10))
  eval <- classify_rf(embed, labels, ntree = 100, seed = 1)
  eval$accuracy
}
```

compare_nmf

Compare Multiple NMF Models

Description

Overlay training curves from multiple NMF models to compare convergence behavior.

Usage

```
compare_nmf(..., labels = NULL, metric = "loss", smooth = TRUE, span = 0.3)
```

Arguments

<code>...</code>	nmf objects to compare
<code>labels</code>	optional character vector of model labels
<code>metric</code>	what to compare: "loss" (default), "sparsity"
<code>smooth</code>	apply smoothing (default TRUE)
<code>span</code>	smoothing span for LOESS (default 0.3)

Value

A `ggplot2` object comparing the requested metric across models.

See Also

[nmf](#), [plot.nmf](#)

Examples

```
library(Matrix)
A <- abs(rsparsematrix(100, 50, 0.2))
model1 <- nmf(A, k = 5, L1 = 0.01)
model2 <- nmf(A, k = 5, L1 = 0.1)
compare_nmf(model1, model2, labels = c("L1=0.01", "L1=0.1"))
```

compute_target

Compute a Target Matrix for Guided NMF

Description

Builds a $k \times n$ target matrix from class labels, suitable for passing as `target_H` to `nmf`. Each sample column is set to its class centroid (optionally ZCA-whitened), so that target regularization steers H toward class-discriminative structure.

Usage

```
compute_target(H, labels, whiten = TRUE)
```

Arguments

<code>H</code>	$k \times n$ embedding matrix (e.g. from an initial NMF fit).
<code>labels</code>	Factor or character/integer vector of class labels (length n). NA entries are left unguided (zero column).
<code>whiten</code>	Logical; apply OAS-shrinkage ZCA whitening to class centroids before broadcasting. Default TRUE.

Details

With positive `target_lambda`, NMF attracts H toward the target (label enrichment). With negative `target_lambda`, NMF uses PROJ_ADV eigenvalue-projected adversarial removal to suppress target-correlated structure (batch removal). See `vignette("guided-nmf")` for mathematical details.

Algorithm:

1. Compute per-class centroids from rows/columns of H .
2. (Optional) Apply Oracle-Approximating Shrinkage (OAS) covariance estimation followed by ZCA whitening to the centroids, ensuring isotropic class structure.
3. Broadcast each sample's class centroid back to a $k \times n$ matrix.

Value

A numeric $k \times n$ matrix. Pass it to `nmf(..., target_H = T, target_lambda = 0.5)` for enrichment or `target_lambda = -0.5` for batch removal.

See Also[nmf](#), [refine](#)**Examples**

```

data(hawaiiibirds)
model <- nmf(hawaiiibirds, k = 4, seed = 42, maxit = 50)
meta <- attr(hawaiiibirds, "metadata_h")

# Build target from class labels
target <- compute_target(model@h, labels = meta$island)

# Enrichment: attract H toward island structure
guided <- nmf(hawaiiibirds, k = 4, seed = 42, maxit = 50,
              target_H = target, target_lambda = 0.5)

# Batch removal: suppress island-correlated structure
removed <- nmf(hawaiiibirds, k = 4, seed = 42, maxit = 50,
               target_H = target, target_lambda = -0.5)

```

consensus_nmf

*Consensus Clustering for NMF***Description**

Run multiple NMF replicates and compute consensus matrix showing co-clustering frequency of samples.

Usage

```

consensus_nmf(
  data,
  k,
  reps = 50,
  method = c("hard", "knn_jaccard"),
  knn = 10,
  seed = NULL,
  threads = 0,
  verbose = FALSE,
  ...
)

```

Arguments

data	input matrix (samples x features for clustering samples)
k	rank of factorization
reps	number of replicates (default 50)

method	consensus method: "hard" for hard cluster assignments (default), or "knn_jaccard" for KNN-based Jaccard overlap of factor loadings
knn	number of nearest neighbors to use for KNN Jaccard method (default 10)
seed	random seed for reproducibility
threads	number of threads for OpenMP parallelization (default 0 = all available)
verbose	print progress information (default FALSE)
...	additional arguments passed to nmf

Details

Consensus clustering runs NMF multiple times with different random initializations.

****Hard clustering method**** (method = "hard"): For each run, samples are clustered based on their dominant factor in W. The consensus matrix C[i,j] gives the proportion of runs where samples i and j were assigned to the same cluster. This is the traditional consensus clustering approach.

****KNN Jaccard method**** (method = "knn_jaccard"): For each run, the k-nearest neighbors of each sample are computed based on factor loadings (W matrix). The consensus matrix C[i,j] is the average Jaccard similarity between the KNN sets of samples i and j across all replicates. This approach is more robust to ambiguous cluster assignments and captures neighborhood structure rather than hard cluster membership.

High consensus values (near 1) indicate stable co-clustering or neighborhood overlap. Intermediate values suggest ambiguous relationships.

The cophenetic correlation coefficient measures cluster stability - higher values (closer to 1) indicate more stable/reproducible clustering.

Value

List with:

- consensus - consensus matrix (samples x samples)
- models - list of fitted nmf objects
- clusters - final cluster assignments
- cophenetic - cophenetic correlation coefficient
- method - consensus method used

See Also

[nmf](#), [plot.consensus_nmf](#), [summary.consensus_nmf](#)

Examples

```
library(Matrix)
A <- rsparsematrix(100, 50, 0.3)

# Traditional hard clustering consensus
cons_hard <- consensus_nmf(A, k = 5, reps = 10, method = "hard", seed = 123)
```

```
# KNN Jaccard consensus (more robust)
cons_knn <- consensus_nmf(A, k = 5, reps = 10, method = "knn_jaccard", knn = 15, seed = 123)

# Plot consensus heatmap
plot(cons_hard)
plot(cons_knn)

# Check cophenetic coefficient (higher = more stable)
print(cons_hard$cophenetic)
print(cons_knn$cophenetic)

# Get cluster assignments
print(table(cons_hard$clusters))
```

cosine

Cosine similarity

Description

Column-by-column Euclidean norm cosine similarity for a matrix, pair of matrices, pair of vectors, or pair of a vector and matrix. Supports sparse matrices.

Usage

```
cosine(x, y = NULL)
```

Arguments

x	matrix or vector of, or coercible to, class "dgCMatrix" or "sparseVector"
y	(optional) matrix or vector of, or coercible to, class "dgCMatrix" or "sparseVector"

Details

This function takes advantage of extremely fast vector operations and is able to handle very large datasets.

cosine applies a Euclidean norm to provide very similar results to Pearson correlation. Note that negative values may be returned due to the use of Euclidean normalization when all associations are largely random.

This function adopts the sparse matrix computational strategy applied by `qlcMatrix::cosSparse`, and extends it to any combination of single and/or pair of sparse matrix and/or dense vector.

Value

dense matrix, vector, or value giving cosine distances

See Also[nmf](#), [bipartiteMatch](#)**Examples**

```
x <- matrix(runif(20), 4, 5)
cosine(x)           # self-similarity: 5x5 matrix
cosine(x, x[,1])    # similarity of columns to first column
```

cross_validate_graph *Cross-validate a factorization network*

Description

Evaluates a factor_net architecture across a grid of hyperparameter combinations using held-out test loss. Each combination is fitted with a fraction of entries masked, and the test loss on those entries determines the optimal configuration.

Usage

```
cross_validate_graph(
  inputs,
  layer_fn,
  params,
  config = factor_config(),
  reps = 3L,
  strategy = c("grid", "random"),
  n_random = 20L,
  seed = 42L,
  verbose = TRUE
)
```

Arguments

inputs	Input node(s) from factor_input(). Can be a single node or a list of nodes for multi-modal.
layer_fn	A function that, given a named list of parameter values, returns the output layer node of the network. Example: function(p) inputs > nmf_layer(k = p\$k, L1 = p\$L1)
params	A named list of parameter vectors to search over. Names should match the arguments expected by layer_fn. Example: list(k = c(5, 10, 20), L1 = c(0, 0.01)).
config	A fn_global_config from factor_config(). The test_fraction, cv_seed, mask_zeros, and patience fields are used for cross-validation. If test_fraction is 0 (default), it is automatically set to 0.1.

reps	Number of CV replicates per parameter combination (each with a different CV mask seed). Default 3.
strategy	Search strategy: "grid" (all combinations) or "random" (sample n_random combinations). Default "grid".
n_random	Number of combinations for random search. Ignored for grid search. Default 20.
seed	Seed for random search sampling and CV mask derivation. Default 42.
verbose	Print progress updates. Default TRUE.

Details

For single-parameter rank selection, pass $k = c(5, 10, 20)$. For multi-parameter search, use `params` to specify named lists of values for each layer and parameter.

Value

A `factor_net_cv` object with components:

results Data frame with columns: param values, rep, test_loss, train_loss, iterations, converged.

summary Data frame with param values, mean_test_loss, se_test_loss, mean_train_loss, ranked by mean_test_loss.

best_params Named list of the best parameter combination.

all_fits List of all fit results (if `keep_fits = TRUE`).

Note

The seed parameter defaults to 42L (deterministic) rather than NULL (random) used by `nmf()` and `svd()`. This ensures reproducible cross-validation grid searches by default. Pass `seed = NULL` for non-deterministic behavior.

See Also

[factor_net](#), [fit](#), [factor_config](#)

Examples

```
library(Matrix)
X <- rsparsematrix(100, 50, 0.1)
inp <- factor_input(X, "X")

# Rank selection
cv <- cross_validate_graph(
  inputs = inp,
  layer_fn = function(p) inp |> nmf_layer(k = p$k),
  params = list(k = c(3, 5, 10, 20)),
  config = factor_config(maxit = 50, seed = 42)
)
print(cv)
cv$best_params # optimal rank
```

```
# Multi-parameter search
cv2 <- cross_validate_graph(
  inputs = inp,
  layer_fn = function(p) inp |> nmf_layer(k = p$k, L1 = p$L1),
  params = list(k = c(5, 10, 20), L1 = c(0, 0.01, 0.1)),
  config = factor_config(maxit = 50, seed = 42),
  reps = 3
)
```

dclust

Divisive clustering

Description

Recursive bipartitioning by rank-2 matrix factorization with an efficient modularity-approximate stopping criteria

Usage

```
dclust(
  A,
  min_samples,
  min_dist = 0,
  tol = 1e-05,
  maxit = 100,
  nonneg = TRUE,
  seed = NULL,
  threads = 0,
  verbose = FALSE
)
```

Arguments

A	matrix of features-by-samples in sparse format (preferred class is "Matrix::dgCMatrix")
min_samples	stopping criteria giving the minimum number of samples permitted in a cluster
min_dist	stopping criteria giving the minimum cosine distance of samples within a cluster to the center of their assigned vs. unassigned cluster. If 0, neither this distance nor cluster centroids will be calculated.
tol	in rank-2 NMF, the correlation distance $(1 - R^2)$ between w across consecutive iterations at which to stop factorization
maxit	maximum number of fitting iterations (default 100)
nonneg	in rank-2 NMF, enforce non-negativity
seed	random seed for rank-2 NMF model initialization
threads	number of threads for OpenMP parallelization (default 0 = all available)
verbose	print progress information (default FALSE)

Details

Divisive clustering is a sensitive and fast method for sample classification. Samples are recursively partitioned into two groups until a stopping criteria is satisfied and prevents successful partitioning.

See [nmf](#) and [bipartition](#) for technical considerations and optimizations relevant to bipartitioning.

Stopping criteria. Two stopping criteria are used to prevent indefinite division of clusters and tune the clustering resolution to a desirable range:

- `min_samples`: Minimum number of samples permitted in a cluster
- `min_dist`: Minimum cosine distance of samples to their cluster center relative to their unassigned cluster center (an approximation of Newman-Girvan modularity)

Newman-Girvan modularity (Q) is an interpretable and widely used measure of modularity for a bipartition. However, it requires the calculation of distance between all within-cluster and between-cluster sample pairs. This is computationally intensive, especially for large sample sets.

`dclust` uses a measure which linearly approximates Newman-Girvan modularity, and simply requires the calculation of distance between all samples in a cluster and both cluster centers (the assigned and unassigned center), which is orders of magnitude faster to compute. Cosine distance is used instead of Euclidean distance since it handles outliers and sparsity well.

A bipartition is rejected if either of the two clusters contains fewer than `min_samples` or if the mean relative cosine distance of the bipartition is less than `min_dist`.

A bipartition will only be attempted if there are more than $2 * \text{min_samples}$ samples in the cluster, meaning that `dist` may not be calculated for some clusters.

Reproducibility. Because rank-2 NMF is approximate and requires random initialization, results may vary slightly across restarts. Therefore, specify a seed to guarantee absolute reproducibility.

Other than setting the seed, reproducibility may be improved by setting `tol` to a smaller number to increase the exactness of each bipartition.

Value

A list of lists corresponding to individual clusters:

- `id` : character sequence of "0" and "1" giving position of clusters along splitting hierarchy
- `samples` : 0-indexed integer indices of samples in the cluster (add 1 for R-style indexing)
- `center` : mean feature expression of all samples in the cluster
- `size` : number of samples in the cluster

Note

`dclust()` uses `A` for the data matrix and scalar `nonneg/tol` parameters (matching [bipartition\(\)](#)). The default `tol = 1e-5` is tighter than [nmf\(\)](#)'s `1e-4` because rank-2 subproblems converge faster and benefit from higher precision.

Author(s)

Zach DeBruine

References

- Schwartz, G. et al. "TooManyCells identifies and visualizes relationships of single-cell clades". Nature Methods (2020).
- Newman, MEJ. "Modularity and community structure in networks". PNAS (2006)
- Kuang, D, Park, H. (2013). "Fast rank-2 nonnegative matrix factorization for hierarchical document clustering." Proc. 19th ACM SIGKDD intl. conf. on Knowledge discovery and data mining.

See Also

[bipartition](#), [nmf](#)

Examples

```
data(USArrests)
A <- as(as.matrix(t(USArrests)), "dgCMatrix")
clusters <- dclust(A, min_samples = 2, min_dist = 0.001)
str(clusters)
```

diagnose_dispersion	<i>Diagnose dispersion mode</i>
---------------------	---------------------------------

Description

Determines whether dispersion should be estimated per-row, per-column, or globally by examining the coefficient of variation of per-row and per-column dispersion estimates.

Usage

```
diagnose_dispersion(data, model, cv_threshold = 0.5, min_mu = 1e-06)
```

Arguments

data	Input matrix (sparse or dense)
model	A fitted NMF model
cv_threshold	CV threshold for declaring structured dispersion. Default 0.5.
min_mu	Floor for predicted values. Default 1e-6.

Details

Computes moment-based dispersion estimates $\hat{\phi} = r^2/\mu^p$ (where p is determined by the distribution) per row and per column. If the coefficient of variation (CV) of per-row estimates exceeds `cv_threshold`, per-row dispersion is recommended; similarly for per-column. If both CVs are low, global dispersion suffices.

Value

A list with:

mode Recommended DispersionMode: "global", "per_row", or "per_col"

global_phi Global dispersion estimate

row_cv CV of per-row dispersion estimates

col_cv CV of per-col dispersion estimates

See Also

[auto_nmf_distribution](#), [nmf](#)

Examples

```
data(aml)
model <- nmf(aml, k = 3, maxit = 10, seed = 1)
disp <- diagnose_dispersion(aml, model)
disp$mode
```

diagnose_zero_inflation

Diagnose zero inflation

Description

Tests whether a dataset has excess zeros beyond what the chosen distribution predicts, and recommends a zero-inflation mode.

Usage

```
diagnose_zero_inflation(data, model, threshold = 0.05)
```

Arguments

data	Input matrix (sparse or dense)
model	A fitted NMF model (any distribution)
threshold	Minimum excess zero fraction to declare ZI. Default 0.05.

Details

Computes the expected number of zeros under the fitted distribution (Poisson approximation: $P(X = 0) \approx e^{-\mu}$), compares to the observed zero count, and recommends ZI if the excess is large.

ZI granularity is determined by whether per-row and per-col excess rates have high variance (suggesting different rows/cols have different ZI levels).

Value

A list with:

excess_zero_rate Fraction of zeros exceeding the expected count

has_zi Logical: TRUE if excess_zero_rate > threshold

zi_mode Recommended mode: "none", "row", or "col"

row_excess Per-row excess zero rates

col_excess Per-col excess zero rates

See Also

[auto_nmf_distribution](#), [nmf](#)

Examples

```
data(aml)
model <- nmf(aml, k = 3, maxit = 10, seed = 1)
zi <- diagnose_zero_inflation(aml, model)
zi$has_zi
```

digits

MNIST Digits Dataset

Description

MNIST handwritten digit dataset containing grayscale images of digits 0-9. Each image is 28x28 pixels flattened to 784 features.

Usage

```
digits
```

Format

A matrix (dense) with pixel intensities. Rows are samples, columns are features.

Details

This is the MNIST dataset, commonly used for benchmarking machine learning algorithms. The data is stored as a dense matrix (not sparse) since handwritten digit images have substantial non-zero content.

For NMF, it's recommended to:

- Normalize pixel values to $[0, 1]$ by dividing by 255
- Transpose to have pixels as rows, samples as columns

- Use a subset for faster experimentation

The true rank is 10 (number of digit classes), though higher ranks may capture stroke variations and writing styles within digits.

Source

MNIST database of handwritten digits. Yann LeCun, Corinna Cortes, Christopher J.C. Burges.
<http://yann.lecun.com/exdb/mnist/>

Examples

```
data(digits)
dim(digits)
model <- nmf(digits, k = 10, maxit = 50)
```

evaluate

Evaluate an NMF model

Description

Calculate loss for an NMF model using the specified loss function, accounting for any masking schemes requested during fitting.

Usage

```
evaluate(x, ...)
```



```
## S4 method for signature 'nmf'
evaluate(
  x,
  data,
  mask = NULL,
  missing_only = FALSE,
  loss = c("mse", "gp"),
  test_fraction = 0,
  test_seed = NULL,
  eval_set = c("all", "test", "train"),
  threads = 0,
  verbose = FALSE,
  ...
)
```

Arguments

<code>x</code>	fitted model, class <code>nmf</code> , generally the result of calling <code>nmf</code> , with models of equal dimensions as data
<code>...</code>	advanced parameters. See Advanced Parameters section.
<code>data</code>	dense or sparse matrix of features in rows and samples in columns. Prefer <code>matrix</code> or <code>Matrix::dgCMatrix</code> , respectively. Also accepts a file path (character string) which will be auto-loaded based on extension.
<code>mask</code>	missing data mask. Accepts: <code>NULL</code> (no masking), <code>"zeros"</code> (mask zeros), <code>"NA"</code> (mask NAs), a <code>dgCMatrix/matrix</code> (custom mask), or <code>list("zeros", <matrix>)</code> to mask zeros and use a custom mask simultaneously.
<code>missing_only</code>	calculate loss only for missing values specified as a matrix in <code>mask</code>
<code>loss</code>	loss function to use: <code>"mse"</code> (Mean Squared Error, default) or <code>"gp"</code> (Generalized Poisson / KL divergence)
<code>test_fraction</code>	fraction of entries to hold out as test set (default 0 = disabled). When > 0 , creates a random mask for test/train split.
<code>test_seed</code>	seed for test set generation. If <code>NULL</code> , attempts to use test mask from model's <code>@misc\$test_mask</code> if available.
<code>eval_set</code>	which set to evaluate: <code>"all"</code> (default), <code>"test"</code> (held-out entries only), or <code>"train"</code> (non-held-out entries only). Only used when <code>test_fraction > 0</code> or test mask exists in model.
<code>threads</code>	number of threads for OpenMP parallelization (default 0 = all available)
<code>verbose</code>	print progress information (default <code>FALSE</code>)

Value

A single numeric value: the loss (MSE or GP/KL divergence) of the model on the data.

See Also

[nmf](#)

Examples

```
library(Matrix)
A <- rsparsematrix(100, 50, 0.1)
model <- nmf(A, 3, seed = 1, maxit = 50, tol = 1e-4)
evaluate(model, A) # MSE
```

export_log	<i>Export training log to CSV</i>
------------	-----------------------------------

Description

Export training log to CSV

Usage

```
export_log(logger, file)
```

Arguments

logger	A training_logger object.
file	Path to write the CSV file.

Value

Invisibly returns the data frame written.

See Also

[training_logger](#), [as.data.frame.training_logger](#)

Examples

```
logger <- training_logger()
# After fitting: export_log(logger, tempfile(fileext = ".csv"))
```

factor_add	<i>Element-wise H addition (skip/residual connection)</i>
------------	---

Description

Creates a node that adds H factors element-wise from multiple branches. All branches must have the same rank k.

Usage

```
factor_add(...)
```

Arguments

...	Two or more fn_node objects with matching ranks.
-----	--

Value

An fn_node of type "add".

See Also

[factor_concat](#), [factor_shared](#), [factor_net](#)

Examples

```
data(aml)
inp <- factor_input(aml)
branch1 <- nmf_layer(inp, k = 5)
branch2 <- nmf_layer(inp, k = 5)
added <- factor_add(branch1, branch2)
```

factor_concat	<i>Concatenate H factors from branches (row-bind)</i>
---------------	---

Description

Creates a node that row-binds the H factors from multiple branches. Allows combining branches with different ranks into a single higher-dimensional representation: $k_{out} = k_1 + k_2 + \dots$

Usage

```
factor_concat(...)
```

Arguments

... Two or more fn_node objects (layer outputs).

Value

An fn_node of type "concat".

See Also

[factor_shared](#), [factor_add](#), [factor_net](#)

Examples

```
data(aml)
inp <- factor_input(aml)
branch1 <- nmf_layer(inp, k = 3)
branch2 <- nmf_layer(inp, k = 5)
combined <- factor_concat(branch1, branch2)
```

factor_condition	<i>Concatenate conditioning metadata to a layer's H</i>
------------------	---

Description

Appends columns from a metadata matrix Z to the H factor before passing downstream. The next layer's W learns to factor out the conditioning variables.

Usage

```
factor_condition(input, Z)
```

Arguments

input	An <code>fn_node</code> (typically an <code>nmf_layer</code> output).
Z	Conditioning matrix ($n \times p$) or ($p \times n$). Will be oriented to match H dimensions.

Value

An `fn_node` of type "condition".

See Also

[nmf_layer](#), [factor_net](#)

Examples

```
data(aml)
inp <- factor_input(aml)
layer1 <- nmf_layer(inp, k = 5)
# Condition on batch metadata (2 batches)
Z <- matrix(rep(c(1, 0, 0, 1), c(60, 75, 60, 75)), nrow = 135, ncol = 2)
conditioned <- factor_condition(layer1, Z)
```

factor_config	<i>Global configuration for a factorization network</i>
---------------	---

Description

Sets network-wide defaults. Layer-level and factor-level settings override these where specified.

Usage

```

factor_config(
  maxit = 100,
  tol = 1e-04,
  loss = c("mse", "gp", "nb", "gamma", "inverse_gaussian", "tweedie"),
  verbose = FALSE,
  seed = NULL,
  threads = 0,
  norm = c("L1", "L2", "none"),
  solver = c("auto", "cholesky", "cd"),
  resource = "auto",
  test_fraction = 0,
  cv_seed = 0L,
  mask_zeros = FALSE,
  patience = 5L,
  ...
)

```

Arguments

<code>maxit</code>	Maximum ALS iterations per layer. Default 100.
<code>tol</code>	Convergence tolerance. Default 1e-4.
<code>loss</code>	Loss function: "mse", "gp", "nb", "gamma", "inverse_gaussian", "tweedie". Default "mse". For robust (Huber/MAE) fitting, use the robust parameter at the layer level instead.
<code>verbose</code>	Print per-iteration diagnostics. Default FALSE.
<code>seed</code>	Random seed. Default NULL (auto).
<code>threads</code>	Number of CPU threads (0 = all). Default 0.
<code>norm</code>	Normalization type: "L1", "L2", "none". Default "L1".
<code>solver</code>	NNLS solver: "auto", "cholesky", "cd". Default "auto".
<code>resource</code>	Resource override: "auto", "cpu", "gpu". Default "auto".
<code>test_fraction</code>	Fraction of entries held out for cross-validation test set. 0 = no CV (standard fit). Default 0.
<code>cv_seed</code>	Separate seed for CV mask (0 = derive from seed).
<code>mask_zeros</code>	If TRUE, only non-zero entries are in the test set (suitable for recommendation systems). If FALSE, all entries including zeros may appear in the test set. Default FALSE.
<code>patience</code>	Number of iterations without test-loss improvement before early stopping during CV. Default 5.
<code>...</code>	Additional arguments applied network-wide. These are forwarded to the underlying <code>nmf</code> or <code>svd</code> call at fit time as lowest-priority defaults (layer-level ... overrides these). Supports distribution tuning (dispersion, theta_init, etc.), IRLS control, solver tuning, and more. See <code>?nmf</code> or <code>?svd</code> for the complete list.

Details**Unsupported combinations:**

- Cholesky solver with non-MSE losses (GP, NB, Gamma, etc.) is not supported. Use solver="cd" or solver="auto".
- Zero-inflation is only available with GP and NB losses at the layer level; it cannot be set globally in factor_config().

Value

An fn_global_config object.

See Also

[factor_net](#), [nmf_layer](#), [fit](#)

Examples

```
# Default config
cfg <- factor_config()

# Poisson NMF with cross-validation
cfg <- factor_config(loss = "gp", test_fraction = 0.1, maxit = 50)
```

factor_input	Create an input node for a factorization network
--------------	--

Description

Wraps a data matrix as a graph input node. The matrix can be dense or sparse (dgCMatrix), or a file path to a .spz file for streaming.

Usage

```
factor_input(data, name = NULL)
```

Arguments

data	A numeric matrix, sparse matrix (dgCMatrix), or character string path to a .spz file for out-of-core streaming NMF.
name	Optional name for the input (used in multi-modal results).

Value

An fn_node object of type "input".

See Also

[nmf_layer](#), [svd_layer](#), [factor_net](#)

Examples

```
data(aml)
inp <- factor_input(aml, name = "aml")
inp
```

factor_net	<i>Compile a factorization network</i>
------------	--

Description

Validates the graph topology, resolves config inheritance, and returns a compiled network ready for `fit()`.

Usage

```
factor_net(inputs, output, config = factor_config())
```

Arguments

inputs	A single <code>fn_node</code> (input) or a list of input nodes.
output	The output <code>fn_node</code> (typically a layer).
config	A <code>fn_global_config</code> from <code>factor_config()</code> . Default uses <code>factor_config()</code> defaults.

Value

A `factor_net` object.

See Also

[fit](#), [factor_config](#), [nmf_layer](#), [svd_layer](#), [cross_validate_graph](#)

Examples

```
data(aml)
inp <- factor_input(aml, "aml")
out <- nmf_layer(inp, k = 5)
net <- factor_net(inp, out, config = factor_config(maxit = 10))
net
```

factor_shared	<i>Shared factorization across multiple inputs (multi-modal)</i>
---------------	--

Description

Creates a node representing shared-H factorization of multiple input matrices. The resulting H is shared across all inputs; each input gets its own W.

Usage

```
factor_shared(...)
```

Arguments

... Two or more `fn_node` objects (inputs or layers).

Details

Execution concatenates inputs row-wise and runs a single NMF: $\text{rbind}(X1, X2, \dots) = \text{rbind}(W1, W2, \dots) * \text{diag}(d) * H$.

Value

An `fn_node` of type "shared".

See Also

[factor_concat](#), [factor_add](#), [factor_net](#)

Examples

```
data(aml)
# Split into two views
inp1 <- factor_input(aml[1:400, ], name = "view1")
inp2 <- factor_input(aml[401:824, ], name = "view2")
shared <- factor_shared(inp1, inp2)
```

fit

*Fit a factorization network***Description**

Executes the compiled graph from `factor_net()`. For single-layer networks, this delegates directly to `nmf()` for maximum performance. Multi-layer networks use R-level alternating least squares.

Usage

```
fit(object, ...)

## S3 method for class 'factor_net'
fit(object, logger = NULL, ...)
```

Arguments

<code>object</code>	A <code>factor_net</code> object from <code>factor_net()</code> .
<code>...</code>	Additional arguments (currently unused).
<code>logger</code>	Optional <code>training_logger</code> from <code>training_logger()</code> for per-iteration diagnostics (loss, norms, classifier metrics).

Value

A `factor_net_result` object. Access per-layer results by name (e.g. `result$L1`). Each layer is a list with components `W`, `d`, `H`, and `iterations`. Additional fields: `n_layers`, `multi_modal`, `total_iterations`, `total_loss`, and `converged`.

See Also

[factor_net](#), [predict.factor_net_result](#), [cross_validate_graph](#)

Examples

```
library(Matrix)
X <- rsparsematrix(100, 50, 0.1)
inp <- factor_input(X, "X")
L1 <- inp |> nmf_layer(k = 5, name = "L1")
net <- factor_net(inputs = inp, output = L1,
                  config = factor_config(maxit = 100, seed = 42))
res <- fit(net)
res$L1$W    # m x k basis matrix
res$L1$H    # k x n coefficient matrix
```

golub*Golub ALL-AML Dataset (Brunet et al. 2004)*

Description

Gene expression data from the landmark Golub et al. (1999) study, consisting of bone marrow samples from acute lymphoblastic leukemia (ALL) and acute myeloid leukemia (AML) patients. This is the 5000 most variable genes subset used in the seminal NMF paper by Brunet et al. (2004, PNAS).

Usage

golub

Format

A dgCMatrix sparse matrix (38 x 5000) containing gene expression values. Rows are patient samples, columns are genes. Cancer type labels are stored as an attribute.

Access metadata via:

- `attr(golub, "cancer_type")` - Factor: "ALL" or "AML"
- `attr(golub, "n_all")` - Number of ALL samples (27)
- `attr(golub, "n_aml")` - Number of AML samples (11)

Details

This dataset is one of the most widely used benchmarks for class discovery and cancer subtype identification. The original study identified gene expression signatures that distinguish ALL from AML.

NMF analysis reveals:

- **True rank = 2:** The two cancer types (ALL vs AML)
- **Optimal rank ~3:** Brunet et al. found rank 3 more informative, potentially capturing ALL subtypes (B-cell vs T-cell)

This dataset has been preprocessed to include only the 5000 most variable genes for computational efficiency, following Brunet et al. (2004).

Source

Golub et al. (1999). "Molecular classification of cancer: class discovery and class prediction by gene expression monitoring." *Science* 286(5439): 531-537.

Brunet et al. (2004). "Metagenes and molecular pattern discovery using matrix factorization." *PNAS* 101(12): 4164-4169.

Data retrieved via NMF package (Gaujoux and Seoighe).

Examples

```
# Load dataset
data(golub)

# Inspect
dim(golub) # 38 samples x 5000 genes
table(attr(golub, "cancer_type")) # 27 ALL, 11 AML

# Run NMF for class discovery
model <- nmf(t(golub), k = 2, maxit = 100) # Transpose: genes x samples

# Try rank 3 as suggested by Brunet et al.
model3 <- nmf(t(golub), k = 3, maxit = 100)
```

gpu_available	<i>Check if GPU acceleration is available</i>
---------------	---

Description

Detects CUDA GPUs at runtime. Result is cached for the session.

Usage

```
gpu_available(force_recheck = FALSE)
```

Arguments

force_recheck Re-probe GPUs even if already cached

Value

logical TRUE if GPU is available

See Also

[gpu_info](#), [nmf](#)

Examples

```
gpu_available()
```

gpu_info	<i>Get GPU device information</i>
----------	-----------------------------------

Description

Get GPU device information

Usage

```
gpu_info()
```

Value

data.frame with GPU device details, or NULL if no GPU

See Also

[gpu_available](#), [nmf](#)

Examples

```
gpu_info()
```

gpu_sparse_matrix-methods	<i>Methods for gpu_sparse_matrix objects</i>
---------------------------	--

Description

S3 methods for the `gpu_sparse_matrix` class returned by [st_read_gpu](#).

Usage

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

```
## S3 method for class 'gpu_sparse_matrix'
dim(x)
```

```
## S3 method for class 'gpu_sparse_matrix'
nrow(x)
```

```
## S3 method for class 'gpu_sparse_matrix'
ncol(x)
```

Arguments

`x` a `gpu_sparse_matrix` object
`...` additional arguments (unused)

Value

`print` Invisibly returns `x`, prints summary to console.
`dim` Integer vector of length 2: `c(nrow, ncol)`.
`nrow` Number of rows (integer).
`ncol` Number of columns (integer).

See Also

[st_read_gpu](#), [st_free_gpu](#)

Examples

```
## Not run:
gpu_mat <- st_read_gpu("data.spz")
print(gpu_mat)
dim(gpu_mat)
nrow(gpu_mat)
ncol(gpu_mat)
st_free_gpu(gpu_mat)

## End(Not run)
```

hawaiiibirds

Hawaii Bird Species Frequency Dataset

Description

Frequency of bird species observations across survey grids in the Hawaiian islands. This dataset contains presence/frequency data for 183 bird species across 1,183 spatial grid cells, along with metadata about survey locations and species characteristics.

Usage

```
hawaiiibirds
```

Format

A `dgCMatrix` sparse matrix (183 x 1,183) with bird species frequency observations. Rows are bird species and columns are survey grid cells. Metadata is stored as attributes.

Access metadata via:

- `attr(hawaiiibirds, "metadata_h")` - Survey grid information (1,183 rows):

grid Character vector of grid cell identifiers
island Factor indicating which Hawaiian island
lat Numeric latitude coordinate
lng Numeric longitude coordinate

- `attr(hawaiiibirds, "metadata_w")` - Species information (183 rows):

species Character vector of species common names
status Factor: "introduced" or "native"
type Factor indicating bird type (e.g., "birds of prey", "seabirds")

Details

This dataset contains bird observation data from the Hawaiian Islands, useful for ecological and biogeographic studies. The data is sparse, with many grid cells containing no observations for most species.

Bird types include:

- Birds of prey
- Coastal and wetland birds
- Land birds
- Seabirds
- Waterbirds

Source

Data originally from RcppML package, derived from Hawaii bird observation surveys.

Examples

```
# Load dataset
data(hawaiiibirds)

# Inspect structure
dim(hawaiiibirds)
table(attr(hawaiiibirds, "metadata_w")$status)
table(attr(hawaiiibirds, "metadata_h")$island)

# Run NMF to identify species co-occurrence patterns
result <- nmf(hawaiiibirds, k = 10, maxit = 50)
```

`movielens`*MovieLens Dataset*

Description

Movie ratings data from the MovieLens 100K dataset, containing user ratings for movies along with genre information.

Usage

```
movielens
```

Format

A `dgCMatrix` sparse matrix (3,867 x 610) containing movie ratings. Rows are movies, columns are users, and values are ratings from 1-5 stars. Most entries are zero (unrated). Genre information is stored as an attribute.

Access genre data via `attr(movielens, "genres")`, which returns an `ngCMatrix` sparse binary matrix (19 x 3,867) indicating genre membership. Rows are genres, columns are movies, and entries are TRUE/FALSE for genre membership.

Details

This is a subset of the MovieLens dataset, widely used for demonstrating collaborative filtering and recommendation systems. The data is highly sparse, as most users have only rated a small fraction of available movies.

The 19 genres include:

- Action, Adventure, Animation, Children
- Comedy, Crime, Documentary, Drama
- Fantasy, Film-Noir, Horror, Musical
- Mystery, Romance, Sci-Fi, Thriller
- War, Western, IMAX

Source

MovieLens 100K Dataset from GroupLens Research. <https://grouplens.org/datasets/movielens/>

Examples

```
# Load dataset
data(movielens)

# Inspect structure
dim(movielens)
dim(attr(movielens, "genres"))
```

```
# Sparsity
Matrix::nnzero(movielens) / prod(dim(movielens))

# Run NMF for collaborative filtering
result <- nmf(movielens, k = 20, maxit = 50)
```

nmf

Non-negative matrix factorization

Description

High-performance NMF of the form $A = wdh$ for large dense or sparse matrices. Supports single-rank fitting or cross-validation across multiple ranks. Returns an `nmf` object or `nmfCrossValidate` `data.frame`.

Usage

```
nmf(
  data,
  k,
  tol = 1e-04,
  maxit = 100,
  L1 = c(0, 0),
  L2 = c(0, 0),
  seed = NULL,
  mask = NULL,
  loss = c("mse", "gp", "nb", "gamma", "inverse_gaussian", "tweedie"),
  nonneg = c(TRUE, TRUE),
  test_fraction = 0,
  verbose = FALSE,
  projective = FALSE,
  symmetric = FALSE,
  zi = c("none", "row", "col"),
  robust = FALSE,
  ...
)
```

Arguments

<code>data</code>	dense or sparse matrix of features in rows and samples in columns. Prefer <code>matrix</code> or <code>Matrix::dgCMatrix</code> , respectively. Also accepts a file path (character string) which will be auto-loaded based on extension.
<code>k</code>	rank (integer), vector of ranks for cross-validation, or "auto" for automatic rank determination.
<code>tol</code>	tolerance of the fit (default 1e-4)

maxit	maximum number of fitting iterations (default 100)
L1	LASSO penalties in the range (0, 1], single value or array of length two for c(w, h)
L2	Ridge penalties greater than zero, single value or array of length two for c(w, h)
seed	initialization control. Accepts: NULL for random init, an integer for reproducible random init, a matrix (m x k) for custom W initialization, a list of matrices for multi-init, or a string: "lanczos", "irlba", "randomized", or "svd" (auto-select) for SVD-based initialization.
mask	missing data mask. Accepts: NULL (no masking), "zeros" (mask zeros), "NA" (mask NAs), a dgCMatrix/matrix (custom mask), or list("zeros", <matrix>) to mask zeros and use a custom mask simultaneously.
loss	loss function: "mse" (default), "gp", "nb", "gamma", "inverse_gaussian", or "tweedie". For robust loss (Huber/MAE), use the robust parameter instead.
nonneg	logical vector of length 2 for c(w, h) specifying non-negativity constraints (default c(TRUE, TRUE)).
test_fraction	fraction of entries to hold out for cross-validation (default 0 = disabled).
verbose	print progress information during fitting (default FALSE)
projective	if TRUE, use projective NMF (default FALSE).
symmetric	if TRUE, use symmetric NMF where $H = W^T$ (default FALSE).
zi	zero-inflation mode: "none" (default), "row", or "col". Requires loss="gp" or loss="nb".
robust	robustness control. FALSE (default), TRUE (Huber delta=1.345), "mae" (near-MAE via very small Huber delta), or a positive numeric Huber delta. Huber loss is quadratic for small residuals and linear for large ones, controlled by delta. TRUE (delta=1.345) provides moderate outlier robustness. A large delta (e.g. 100) approaches standard MSE; a small delta (e.g. 0.01) approaches MAE. "mae" is shorthand for delta=1e-4 (effectively L1 loss).
...	advanced parameters. See Advanced Parameters section.

Details

This fast NMF implementation decomposes a matrix A into lower-rank non-negative matrices w and h , with columns of w and rows of h scaled to sum to 1 via multiplication by a diagonal, d :

$$A = wd h$$

The scaling diagonal ensures convex L1 regularization, consistent factor scalings regardless of random initialization, and model symmetry in factorizations of symmetric matrices.

The factorization model is randomly initialized. w and h are updated by alternating least squares.

RcppML achieves high performance using the Eigen C++ linear algebra library, OpenMP parallelization, a dedicated Rcpp sparse matrix class, and fast sequential coordinate descent non-negative least squares initialized by Cholesky least squares solutions.

Sparse optimization is automatically applied if the input matrix `A` is a sparse matrix (i.e. `Matrix::dgCMatrix`). There are also specialized back-ends for symmetric, rank-1, and rank-2 factorizations.

L1 penalization can be used for increasing the sparsity of factors and assisting interpretability. Penalty values should range from 0 to 1, where 1 gives complete sparsity.

Set `options(RcppML.verbose = TRUE)` to print model tolerances to the console after each iteration.

Parallelization is applied with OpenMP using the number of threads in `getOption("RcppML.threads")` and set by `option(RcppML.threads = 0)`, for example. 0 corresponds to all threads, let OpenMP decide.

Value

When `k` is a single integer: an S4 object of class `nmf` with slots:

`w` feature factor matrix, $m \times k$

`d` scaling diagonal vector of length `k` (descending order after sorting)

`h` sample factor matrix, $k \times n$

`misc` list containing `tol` (final tolerance), `iter` (iteration count), `loss` (final loss value), `loss_type` (loss function used), and `runtime` (seconds).

When `k` is a vector: a data.frame of class `nmfCrossValidate` with columns `k`, `rep`, `train_loss`, `test_loss`, and `best_iter`.

Cross-Validation

When `k` is a vector, the function performs cross-validation to find the optimal rank:

- A fraction (`test_fraction`) of entries are held out for validation
- Models are fit for each rank in `k` using only training data
- Test loss is computed on held-out entries
- Returns a data.frame with `k`, `rep`, `train_mse`, `test_mse`, and `best_iter` for each rank/replicate
- Use `plot()` on the result to visualize loss vs rank

Loss Functions

The `loss` parameter controls the objective function:

- `"mse"`: Mean Squared Error (default). Standard Frobenius norm minimization.
- `"gp"`: Generalized Poisson. For overdispersed count data (e.g., scRNA-seq). Use `dispersion` to control per-row or global overdispersion estimation. With `dispersion="none"`, equivalent to KL divergence (Poisson model).
- `"nb"`: Negative Binomial. Quadratic variance-mean relationship. Standard for scRNA-seq data with overdispersion. Size parameter (`r`) estimated per-row or globally.
- `"gamma"`: Gamma distribution. Variance proportional to μ^2 . For positive continuous data. Dispersion (`phi`) estimated via MoM Pearson estimator.
- `"inverse_gaussian"`: Inverse Gaussian. Variance proportional to μ^3 . For positive continuous data with heavy right tails.

- "tweedie": Tweedie distribution. Variance proportional to μ^p where p is set via `tweedie_power` (default 1.5). Interpolates between Poisson ($p=1$), Gamma ($p=2$), and Inverse Gaussian ($p=3$).

For robust fitting (equivalent to Huber or MAE loss), use the `robust` parameter instead of a separate loss function. Setting `robust = TRUE` applies Huber-weighted IRLS with $\delta=1.345$; `robust = "mae"` approximates mean absolute error.

Non-MSE losses use Iteratively Reweighted Least Squares (IRLS) which may be slower but provides better fits for count data (GP, NB) and positive continuous data (Gamma, Inverse Gaussian).

Advanced Parameters (via ...)

The following parameters can be passed via ...:

Regularization:

`L21` Group sparsity penalty, single value or $c(w, h)$ (default $c(0, 0)$)
`angular` Angular decorrelation penalty, $c(w, h)$ (default $c(0, 0)$)
`upper_bound` Box constraint on factors, $c(w, h)$ (default $c(0, 0) = \text{no bound}$)
`graph_W, graph_H` Sparse graph Laplacian matrices for feature/sample regularization
`graph_lambda` Graph regularization strength, $c(w, h)$ (default $c(0, 0)$)
`target_H` Target matrix ($k \times n$) for H-side regularization. Steers H toward a desired structure during fitting. See [compute_target](#).
`target_lambda` Target regularization strength, single value or $c(w, h)$ (default 0). Positive values attract H toward the target (label enrichment); negative values use PROJ_ADV eigenvalue-projected adversarial removal to suppress target-correlated structure (batch removal). See `vignette("guided-nmf")` for details.

Distribution Tuning:

`dispersion` Dispersion mode for GP loss: "per_row", "per_col", "global", "none"
`theta_init, theta_max, theta_min` GP theta bounds
`nb_size_init, nb_size_max, nb_size_min` NB dispersion bounds
`gamma_phi_init, gamma_phi_max, gamma_phi_min` Gamma/IG dispersion bounds
`tweedie_power` Tweedie variance power (default 1.5)
`irls_max_iter, irls_tol` IRLS convergence parameters

Zero-Inflation:

`zi_em_iters` EM iterations per NMF step (default 1)

Solver:

`solver` NNLS solver: "auto", "cholesky", "cd"
`cd_tol` CD convergence tolerance (default $1e-8$)
`cd_maxit` CD max iterations (default 100)
`h_init` Custom initial H matrix

Cross-Validation:

`cv_seed` Separate seed(s) for CV holdout pattern
`patience` Early stopping patience (default 5)
`cv_k_range` Auto-rank search range (default `c(2, 50)`)
`track_train_loss` Track training loss in CV (default TRUE)

Resources & Output:

`threads` OpenMP threads (default 0 = all)
`resource` Compute backend: "auto", "cpu", "gpu"
`norm` Factor normalization: "L1", "L2", "none"
`sort_model` Sort factors by diagonal (default TRUE)

Streaming:

`streaming` SPZ streaming mode: "auto", TRUE, FALSE
`panel_cols` Panel size for streaming (default 0 = auto)
`dispatch` StreamPress dispatch override

Callbacks:

`on_iteration` Per-iteration callback function
`profile` Enable timing profiling (default FALSE)

Compute Resources (CPU vs GPU)

By default (`resource = "auto"`), RcppML auto-detects available hardware. All features are fully supported on CPU (the default backend). GPU acceleration (when compiled with CUDA support) accelerates sparse and dense NMF. GPU is experimental and falls back to CPU automatically if unavailable.

Methods

S4 methods available for the `nmf` class:

- `predict`: project an NMF model (or partial model) onto new samples
- `evaluate`: calculate mean squared error loss of an NMF model
- `summary`: `data.frame` giving fractional, total, or mean representation of factors in samples or features grouped by some criteria
- `align`: find an ordering of factors in one `nmf` model that best matches those in another `nmf` model
- `prod`: compute the dense approximation of input data
- `sparsity`: compute the sparsity of each factor in w and h
- `subset`: subset, reorder, select, or extract factors (same as `[]`)
- generics such as `dim`, `dimnames`, `t`, `show`, `head`

Author(s)

Zach DeBruine

References

DeBruine, ZJ, Melcher, K, and Triche, TJ. (2021). "High-performance non-negative matrix factorization for large single-cell data." BioRxiv.

See Also

[predict](#), [mse](#), [nnls](#), [align](#), [summary](#), [nmf-method](#)

Examples

```
# basic NMF
model <- nmf(Matrix::rsparsematrix(1000, 100, 0.1), k = 10)

# cross-validation to find optimal rank
sim <- simulateNMF(200, 80, k = 5, noise = 3.0, seed = 42)
cv <- nmf(sim$A, k = 2:10, test_fraction = 0.05, cv_seed = 1:3,
          tol = 1e-5, maxit = 200)
plot(cv)
optimal_k <- cv$k[which.min(cv$test_mse)]

# fit final model with optimal rank
model <- nmf(sim$A, optimal_k)
```

nmf-class

nmf S4 Class

Description

The S4 class for NMF model results.

Slots

w feature factor matrix

d scaling diagonal vector

h sample factor matrix

misc list containing optional components including tol (tolerance), iter (iterations), loss (final loss value), loss_type (loss function used), runtime (in seconds), w_init (initial w matrix), test_mask (CV test set), test_seed (CV seed), test_fraction (CV holdout fraction), train_loss (CV training loss), test_loss (CV test loss), and best_iter (CV best iteration)

See Also

[nmf](#), [evaluate](#), [align](#), [predict](#), [nmf-method](#)

nmf_layer

*Create an NMF factorization layer***Description**

Creates a non-negative matrix factorization layer that decomposes its input into $W * \text{diag}(d) * H$ with non-negativity constraints.

Usage

```
nmf_layer(
  input,
  k,
  L1 = 0,
  L2 = 0,
  L21 = 0,
  angular = 0,
  upper_bound = 0,
  mask = NULL,
  zi = c("none", "row", "col"),
  projective = FALSE,
  symmetric = FALSE,
  robust = FALSE,
  W = NULL,
  H = NULL,
  name = NULL,
  ...
)
```

Arguments

input	An <code>fn_node</code> (input, shared, condition, or another layer).
k	Factorization rank.
L1	L1 penalty (shared by W and H unless overridden). Default 0.
L2	L2 penalty. Default 0.
L21	Group sparsity penalty. Default 0.
angular	Orthogonality penalty. Default 0.
upper_bound	Box constraint. Default 0.
mask	Masking mode: NULL (none), "zeros", "NA", or a sparse mask matrix. See nmf for details.
zi	Zero-inflation mode: "none", "row", or "col". Requires <code>loss = "gp"</code> or <code>"nb"</code> in <code>factor_config()</code> . Default "none".
projective	Use projective NMF (W is reused as H). Default FALSE.
symmetric	Use symmetric NMF (W == H). Default FALSE.

<code>robust</code>	Robustness control: FALSE (default, no robustness), TRUE (Huber delta=1.345), "mae" (near-MAE, delta=1e-4), or a positive numeric Huber delta. See nmf for details.
<code>W</code>	Optional <code>W()</code> config to override W-specific settings.
<code>H</code>	Optional <code>H()</code> config to override H-specific settings.
<code>name</code>	Optional layer name (for results access).
<code>...</code>	Additional arguments forwarded to nmf at fit time. Supports all advanced parameters: distribution tuning (dispersion, theta_init, etc.), IRLS control (irls_max_iter, irls_tol), solver tuning (cd_tol, cd_maxit), streaming (streaming, panel_cols), callbacks (on_iteration), and more. See <code>?nmf</code> for the complete list.

Details

When used with `|>`, the input node is the first argument: `x |> nmf_layer(k = 64)`.

Layer-level regularization parameters (L1, L2, etc.) apply to both `W` and `H` unless overridden by `W()` or `H()`.

Value

An `fn_node` of type "nmf_layer".

See Also

[svd_layer](#), [factor_net](#), [factor_input](#), [nmf](#)

Examples

```
data(aml)
inp <- factor_input(aml)
layer <- nmf_layer(inp, k = 5)

# With zero-inflation and distribution-specific tuning
layer_gp <- nmf_layer(inp, k = 5, zi = "row",
                      dispersion = "per_col", theta_init = 0.5)
```

Description

Project a factor matrix onto new data to solve for the complementary matrix. Given W and A , find H such that $\|WH - A\|^2$ is minimized. Alternatively, given H and A , find W such that $\|WH - A\|^2$ is minimized.

Usage

```
nnls(
  w = NULL,
  h = NULL,
  A,
  L1 = c(0, 0),
  L2 = c(0, 0),
  loss = "mse",
  upper_bound = c(0, 0),
  nonneg = c(TRUE, TRUE),
  threads = 0,
  verbose = FALSE,
  ...
)
```

Arguments

w	factor model matrix (n_features x k) for solving H. Set to NULL to solve for W.
h	factor model matrix (k x n_samples) for solving W. Set to NULL to solve for H.
A	data matrix. Can be dense (matrix) or sparse (dgCMatrix).
L1	L1/LASSO penalty, length 1 or 2. Range [0, 1).
L2	Ridge penalty, length 1 or 2. Range [0, Inf).
loss	loss function: "mse" (default) or others via NMF dispatch.
upper_bound	maximum value in solution, length 1 or 2. 0 = no bound.
nonneg	non-negativity constraints, length 1 or 2.
threads	number of threads for OpenMP parallelization (0 = all available).
verbose	print progress information.
...	advanced parameters. See Advanced Parameters section.

Details

This function solves NNLS projection problems with full flexibility.

Projection Modes:

- w provided, h = NULL: Solve for H given W and A (standard projection)
- h provided, w = NULL: Solve for W given H and A (transpose projection)
- Exactly one of w or h must be NULL

Value

matrix of dimension (k x n_samples) for H or (n_features x k) for W

Advanced Parameters (via ...)

L21 L2,1 group sparsity penalty, length 1 or 2 (default $c(0, 0)$)
 angular Angular decorrelation penalty, length 1 or 2 (default $c(0, 0)$)
 cd_maxit Max coordinate descent iterations (default 100)
 cd_tol CD stopping tolerance (default $1e-8$)
 warm_start Optional initial solution matrix
 dispersion Dispersion estimation mode: "per_row" (default) or "global"
 theta_init Initial GP theta (default 0.1)
 theta_max Maximum GP theta (default 5.0)
 theta_min Minimum GP theta (default 0.0)
 nb_size_init Initial NB size parameter (default 10.0)
 nb_size_max Maximum NB size (default $1e6$)
 nb_size_min Minimum NB size (default 0.01)
 gamma_phi_init Initial Gamma shape parameter (default 1.0)
 gamma_phi_max Maximum Gamma shape (default $1e4$)
 gamma_phi_min Minimum Gamma shape (default $1e-6$)
 tweedie_power Tweedie variance power (default 1.5)
 irls_max_iter Maximum IRLS iterations per solve (default 5)
 irls_tol IRLS convergence tolerance (default $1e-4$)

Target Regularization

When `target_H` and `target_lambda` are provided (via ...), `nnls()` routes the solve through a single NMF iteration internally. Positive `target_lambda` attracts the solution toward `target_H` (label enrichment). Negative `target_lambda` uses eigenvalue-projected adversarial removal (PROJ_ADV) to suppress target-correlated structure (batch removal). See `vignette("guided-nmf")` for details.

Author(s)

Zach DeBruine

References

DeBruine, ZJ, Melcher, K, and Triche, TJ. (2021). "High-performance non-negative matrix factorization for large single-cell data." *BioRxiv*.

Franc, VC, Hlavac, VC, and Navara, M. (2005). "Sequential Coordinate-Wise Algorithm for the Non-negative Least Squares Problem."

See Also

[nmf](#)

Examples

```
# Generate synthetic NMF problem
set.seed(123)
w <- matrix(runif(100), 20, 5) # 20 features, 5 factors
h_true <- matrix(runif(50), 5, 10) # 5 factors, 10 samples
A <- w %*% h_true + matrix(rnorm(200, 0, 0.1), 20, 10) # Add noise

# Project W onto new data to find H
h_recovered <- nnls(w = w, A = A)
cor(as.vector(h_true), as.vector(h_recovered))

# With L1 penalty for sparse H
h_sparse <- nnls(w = w, A = A, L1 = c(0, 0.1))
```

olivetti

Olivetti Faces Dataset

Description

Grayscale face images from AT&T Laboratories Cambridge. This dataset contains 400 face images (64x64 pixels) from 40 subjects, with 10 images per subject showing different poses, expressions, and lighting conditions.

Usage

```
olivetti
```

Format

A dgCMatrix sparse matrix (400 x 4096) containing grayscale face images. Each row is a flattened 64x64 pixel image with values in $[0, 1]$. Subject labels are stored as an attribute.

Access metadata via:

- `attr(olivetti, "subject")` - Factor indicating which of 40 subjects
- `attr(olivetti, "image_shape")` - `c(64, 64)` image dimensions
- `attr(olivetti, "n_subjects")` - Number of subjects (40)
- `attr(olivetti, "images_per_subject")` - Images per subject (10)

Details

This dataset is commonly used for face recognition, clustering, and dimensionality reduction benchmarks. The images show variation in:

- Facial expression (smiling, neutral, etc.)
- Head pose (left, right, up, down)
- Lighting conditions

- Accessories (glasses on/off in some cases)

The true rank for NMF is 40 (number of subjects), though lower ranks may capture common facial features and higher ranks may distinguish expression and pose variations within subjects.

To reshape a row back to an image: `matrix(olivetti[i,], nrow=64, ncol=64, byrow=TRUE)`

Source

AT&T Laboratories Cambridge (formerly Olivetti Research Laboratory). <https://www.cl.cam.ac.uk/research/dtg/attarchive/facedatabase.html>

Original images 92x112, downsampled to 64x64 in sklearn version.

Examples

```
# Load dataset
data(olivetti)

# Inspect
dim(olivetti) # 400 x 4096
table(attr(olivetti, "subject")) # 10 images per subject

# Visualize first face
face_img <- matrix(olivetti[1,], nrow=64, ncol=64, byrow=TRUE)
image(t(face_img)[,64:1], col=grey.colors(256))

# Run NMF to discover face components (small k for speed)
model <- nmf(t(olivetti), k = 5, maxit = 10)
```

pbmc3k

PBMC 3k Single-Cell RNA-seq Dataset (StreamPress Compressed)

Description

The full 10x Genomics PBMC 3k single-cell RNA-seq dataset with Seurat cell type annotations, shipped as StreamPress-compressed raw bytes. Contains 13,714 genes across 2,638 cells with 9 annotated cell types.

Usage

pbmc3k

Format

A raw vector containing StreamPress (.spz) compressed bytes. To obtain the sparse matrix, write the bytes to a temporary file and read with `st_read`:

```
data(pbmc3k)
tmp <- tempfile(fileext = ".spz")
writeBin(pbmc3k, tmp)
counts <- st_read(tmp)
# counts is a dgMatrix: 13,714 genes x 2,638 cells
```

Cell type annotations are embedded in the .spz file as column (var) metadata:

```
cell_types <- st_read_var(tmp)$cell_type
table(cell_types)
```

Details

The underlying matrix is a `dgMatrix` with 13,714 rows (genes) and 2,638 columns (cells), containing 2,238,732 non-zero entries (integer UMI counts). Cell type annotations (9 types: Naive CD4 T, Memory CD4 T, CD14+ Mono, B, CD8 T, FCGR3A+ Mono, NK, DC, Platelet) were obtained from the Seurat `pbmc3k.final` reference object via the `SeuratData` package and stored as StreamPress column metadata.

This dataset is commonly used for demonstrating single-cell analysis workflows including distribution-aware NMF and zero-inflation diagnostics.

Source

10x Genomics PBMC 3k dataset, processed with Seurat (`SeuratData::pbmc3k.final`).

Examples

```
# Load the compressed bytes
data(pbmc3k)

# Decompress to sparse matrix
tmp <- tempfile(fileext = ".spz")
writeBin(pbmc3k, tmp)
counts <- st_read(tmp)
dim(counts) # 13714 x 2638

# Access cell type annotations
cell_types <- st_read_var(tmp)$cell_type
table(cell_types)
```

pca	<i>PCA (centered SVD)</i>
-----	---------------------------

Description

Convenience wrapper around [svd](#) with center = TRUE.

Usage

```
pca(A, k = 10, ...)
```

Arguments

A	Input matrix. May be dense (<code>matrix</code>), sparse (<code>dgCMatrix</code>), or a path to a <code>.spz</code> file for out-of-core streaming SVD.
k	Number of factors (rank). Use "auto" for automatic rank selection via cross-validation. Default: 10.
...	Additional arguments passed to svd .

Value

An S4 object of class `svd` (see [svd](#)).

See Also

[svd](#)

Examples

```
library(Matrix)
data(aml)
result <- pca(aml, k = 5)
result
```

plot.consensus_nmf	<i>Plot Consensus Matrix Heatmap</i>
--------------------	--------------------------------------

Description

Plot Consensus Matrix Heatmap

Usage

```
## S3 method for class 'consensus_nmf'
plot(
  x,
  cluster_rows = TRUE,
  cluster_cols = TRUE,
  show_clusters = TRUE,
  color_palette = c("white", "#fee5d9", "#fcae91", "#fb6a4a", "#de2d26", "#a50f15"),
  interactive = FALSE,
  ...
)
```

Arguments

<code>x</code>	consensus_nmf object
<code>cluster_rows</code>	whether to reorder rows by hierarchical clustering (default TRUE)
<code>cluster_cols</code>	whether to reorder columns (default TRUE, same as rows)
<code>show_clusters</code>	whether to show cluster assignments as sidebar (default TRUE)
<code>color_palette</code>	color palette name or vector of colors
<code>interactive</code>	whether to make interactive plotly heatmap (default FALSE)
<code>...</code>	additional arguments (unused)

Value

A ggplot2 object (or plotly object if `interactive = TRUE`) showing the consensus heatmap.

See Also

[consensus_nmf](#), [summary.consensus_nmf](#)

Examples

```
library(Matrix)
A <- rsparsematrix(50, 30, 0.3)
cons <- consensus_nmf(A, k = 3, reps = 5, seed = 42)
if (requireNamespace("ggplot2", quietly = TRUE)) {
  plot(cons)
}
```

plot.dclust	<i>Plot divisive clustering hierarchy</i>
-------------	---

Description

Reconstructs and plots the binary splitting tree from a dclust result. Each cluster's binary path ID encodes its position in the hierarchy (e.g., "01" = root->left->right). If labels are provided, a stacked composition bar is drawn below each leaf showing label proportions.

Usage

```
## S3 method for class 'dclust'
plot(
  x,
  labels = NULL,
  palette = NULL,
  main = "Divisive Clustering Hierarchy",
  ...
)
```

Arguments

x	a dclust object (list of clusters with binary path IDs)
labels	optional character or factor vector of class labels, one per sample in the original data matrix passed to dclust
palette	optional named character vector mapping label levels to colors. If NULL, generated automatically.
main	plot title
...	additional arguments passed to plot.dendrogram

Value

x (invisibly)

See Also

[dclust](#)

plot.nmf	<i>Plot NMF Training History and Diagnostics</i>
----------	--

Description

TensorBoard-style visualization of NMF training dynamics, convergence analysis, and factor diagnostics. Provides multiple plot types for comprehensive model analysis.

Usage

```
## S3 method for class 'nmf'
plot(
  x,
  type = c("loss", "convergence", "regularization", "sparsity"),
  smooth = TRUE,
  span = 0.3,
  log_scale = FALSE,
  interactive = FALSE,
  theme = "classic",
  ...
)
```

Arguments

x	object of class "nmf"
type	plot type: - "loss": Loss components over iterations (default) - "convergence": Log-scale loss convergence - "regularization": Regularization penalty contributions - "sparsity": Factor sparsity patterns
smooth	apply smoothing (LOESS) for noisy curves (default TRUE)
span	smoothing span for LOESS (default 0.3)
log_scale	use log scale for y-axis (default FALSE, auto TRUE for "convergence")
interactive	create interactive plotly plot (default FALSE)
theme	ggplot2 theme: "classic", "minimal", "dark" (default "classic")
...	additional arguments passed to specific plotting functions

Value

ggplot2 or plotly object

See Also

[nmf](#), [compare_nmf](#)

Examples

```
# Basic loss plot
model <- nmf(hawaiiibirds, k = 10)
plot(model)

# Convergence analysis
plot(model, type = "convergence")

# Interactive plot
plot(model, type = "loss", interactive = TRUE)

# Compare multiple runs
models <- replicate(5, nmf(hawaiiibirds, k = 10), simplify = FALSE)
plot(models[[1]], type = "sparsity")
```

plot.nmfCrossValidate *Plot Cross-Validation Results*

Description

Visualize NMF cross-validation results showing test (and optionally train) loss across candidate ranks. Useful for selecting the optimal factorization rank.

Usage

```
## S3 method for class 'nmfCrossValidate'
plot(x, show_train = NULL, point_size = 3, interactive = FALSE, ...)
```

Arguments

x	object of class nmfCrossValidate (a data.frame from nmf(k = 2:10, test_fraction = 0.1))
show_train	logical, overlay train loss on the plot (default TRUE if train_mse column is available and non-NA)
point_size	size of data points (default 3)
interactive	create interactive plotly plot (default FALSE)
...	additional arguments (unused)

Value

ggplot2 or plotly object

See Also

[nmf](#)

Examples

```
library(Matrix)
A <- abs(rsparsematrix(50, 30, 0.3))
cv_result <- nmf(A, k = 2:6, test_fraction = 0.1, cv_seed = 1, maxit = 20)
plot(cv_result)
```

```
plot.training_logger
```

Plot training log

Description

Produces a multi-panel plot of loss, classifier metrics, and factor norms over training iterations.

Usage

```
## S3 method for class 'training_logger'
plot(x, ...)
```

Arguments

```
x          A training_logger object.
...        Ignored.
```

Value

Invisibly returns NULL. Called for its side effect (plotting).

See Also

[training_logger](#)

```
predict.factor_net_result
```

Project new data through a trained factor network

Description

For single-layer results, projects new samples using the trained W and d . For multi-layer results, chains through each layer.

Usage

```
## S3 method for class 'factor_net_result'
predict(object, newdata, ...)
```

Arguments

object	A factor_net_result from fit().
newdata	A matrix (features x samples) to project.
...	Additional arguments (currently unused).

Value

For single-layer: a matrix (k x n_new). For multi-layer: a list of H matrices, one per layer.

See Also

[fit](#), [factor_net](#)

print.factor_net	<i>Print a factor_net</i>
------------------	---------------------------

Description

Print a factor_net

Usage

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

Arguments

x	A factor_net object.
...	Additional arguments (unused).

Value

Invisibly returns x.

See Also

[factor_net](#), [fit](#)

```
print.factor_net_cv      Print a factor_net_cv result
```

Description

Print a factor_net_cv result

Usage

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

Arguments

x	A factor_net_cv object.
...	Additional arguments (unused).

Value

Invisibly returns x.

See Also

[cross_validate_graph](#), [factor_net](#)

```
print.factor_net_result  
      Print a factor_net_result
```

Description

Print a factor_net_result

Usage

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

Arguments

x	A factor_net_result object.
...	Additional arguments (unused).

Value

Invisibly returns x.

See Also

[factor_net](#), [fit](#), [summary.factor_net_result](#)

```
print.fn_classifier_eval
```

Print a classifier evaluation result

Description

Displays a human-readable summary of classification metrics including accuracy, macro/weighted F1, AUC, and per-class precision/recall.

Usage

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

Arguments

x	An <code>fn_classifier_eval</code> object returned by classify_embedding , classify_logistic , or classify_rf .
...	Additional arguments (unused).

Value

Invisibly returns x.

See Also

[summary.fn_classifier_eval](#), [classify_embedding](#)

```
print.fn_factor_config
```

Print an fn_factor_config

Description

Print an `fn_factor_config`

Usage

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

Arguments

x	An <code>fn_factor_config</code> object.
...	Additional arguments (unused).

Value

Invisibly returns x.

See Also

[W](#), [H](#), [factor_config](#)

`print.fn_global_config`

Print an `fn_global_config`

Description

Print an `fn_global_config`

Usage

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

Arguments

x	An <code>fn_global_config</code> object.
...	Additional arguments (unused).

Value

Invisibly returns x.

See Also

[factor_config](#), [factor_net](#)

print.fn_node	<i>Print an fn_node</i>
---------------	-------------------------

Description

Print an fn_node

Usage

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

Arguments

x	An fn_node object.
...	Additional arguments (unused).

Value

Invisibly returns x.

See Also

[factor_input](#), [nmf_layer](#), [factor_net](#)

print.nmf_assessment	<i>Print method for nmf_assessment objects</i>
----------------------	--

Description

Print method for nmf_assessment objects

Usage

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

Arguments

x	an nmf_assessment object
...	additional arguments (unused)

Value

Invisibly returns x

```
print.training_logger
```

Print a training log

Description

Print a training log

Usage

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

Arguments

x	A training_logger object.
...	Additional arguments (unused).

Value

Invisibly returns x.

See Also

[training_logger](#)

```
refine
```

Refine an NMF Model Using Label-Guided Correction

Description

Post-hoc correction of an NMF embedding to improve class separation or remove batch effects, optionally followed by W-refit cycles that propagate the corrected H back through the model.

Usage

```
refine(  
  x,  
  data = NULL,  
  labels,  
  batch = NULL,  
  lambda = 0.8,  
  cycles = 0L,  
  nonneg = TRUE,  
  whiten = TRUE  
)
```

Arguments

<code>x</code>	An NMF model (S4 object of class <code>nmf</code>) or a $k \times n$ embedding matrix.
<code>data</code>	Original data matrix (required when <code>cycles > 0</code> or <code>batch</code> is supplied with <code>cycles > 0</code>). Must be the same matrix used to fit <code>x</code> .
<code>labels</code>	Factor or character/integer vector of class labels (length <code>n</code>).
<code>batch</code>	Optional factor of batch labels (length <code>n</code>) for batch removal. When supplied, batch-correlated structure is suppressed using the PROJ_ADV method (eigenvalue-projected adversarial removal). See <code>vignette("guided-nmf")</code> for details.
<code>lambda</code>	Correction strength in $[0, 1]$. Default 0.8. Controls both label enrichment (positive direction) and batch removal strength (when <code>batch</code> is supplied).
<code>cycles</code>	Number of W-refit cycles. 0 = post-hoc only (default). Each cycle: refit W from corrected H, refit H from new W, re-correct H.
<code>nonneg</code>	Enforce non-negativity on refitted factors. Default TRUE.
<code>whiten</code>	Apply OAS-ZCA whitening to class centroids. Default TRUE.

Details

The correction proceeds in two stages:

Stage 1: Post-hoc centroid correction (always performed)

Computes a target matrix from `labels` via `compute_target`, then shifts each sample's embedding toward its class centroid:

$$H_{corrected} = H + \lambda \cdot T$$

where `T` is the target matrix (class centroid shifts, optionally whitened).

Stage 1b: Batch removal (when `batch` is supplied)

Uses PROJ_ADV (Projected Adversarial) method: computes a batch target from batch labels with negative `target_lambda`, which subtracts the batch Gram matrix from the NNLS Gram matrix, eigendecomposes, and clips negative eigenvalues. This suppresses batch-correlated directions while preserving all other structure.

Stage 2: W-refit cycles (when `cycles > 0`)

Iteratively refits the model:

1. Solve for W given corrected H: $W = \arg \min \|A - WdH_c\|_F^2$
2. Solve for H given new W: $H = \arg \min \|A - W_{new}dH\|_F^2$
3. Re-apply centroid correction to the new H

Value

If `x` is an `nmf` object, returns an updated `nmf` object. If `x` is a matrix, returns a corrected $k \times n$ matrix.

See Also

`compute_target`, `nmf`

Examples

```
data(hawaiiibirds)
model <- nmf(hawaiiibirds, k = 4, seed = 42, maxit = 50)
meta <- attr(hawaiiibirds, "metadata_h")

# Post-hoc correction only (cycles = 0)
corrected <- refine(model, labels = meta$island, lambda = 0.8)

# W-refit cycles: propagate correction back through the model
refined <- refine(model, data = hawaiiibirds, labels = meta$island,
                  lambda = 0.8, cycles = 3)
```

score_test_distribution

Score-test distribution diagnostic

Description

Given a baseline MSE-fitted NMF model and the original data, computes score-test statistics for the power-variance family ($V(\mu) = \mu^P$) to determine the best-fitting distribution without refitting. Optionally tests for NB overdispersion.

Usage

```
score_test_distribution(
  data,
  model,
  powers = c(0, 1, 2, 3),
  test_nb = TRUE,
  min_mu = 1e-06
)
```

Arguments

data	Original data matrix (sparse or dense)
model	A fitted NMF object (from <code>nmf()</code> with <code>loss="mse"</code>)
powers	Numeric vector of variance powers to test. Default <code>c(0, 1, 2, 3)</code> covers Gaussian, Poisson, Gamma, Inverse Gaussian.
test_nb	Logical; if TRUE and data is integer-valued, also test the NB overdispersion diagnostic.
min_mu	Floor for predicted values to avoid division by zero. Default 1e-6.

Details

The score test statistic for variance power p is:

$$T_p = \text{mean} \left(\frac{r_{ij}^2}{\mu_{ij}^p} - 1 \right)$$

where $r_{ij} = x_{ij} - \mu_{ij}$ are residuals and $\mu_{ij} = (WH)_{ij}$ are predicted values.

Under the correct model, $E[T_p] = 0$. The power minimizing $|T_p|$ best matches the observed variance-mean relationship.

The NB diagnostic tests for quadratic overdispersion:

$$T_{NB} = \text{mean} \left(\frac{r_{ij}^2 - \mu_{ij}}{\mu_{ij}^2} \right)$$

If $T_{NB} > 0.1$, there is substantial overdispersion beyond Poisson, suggesting NB may be preferable to GP.

Value

A list with:

scores Data frame with columns power, T_stat, abs_T, distribution (label)

best_power Numeric: the power p with smallest $|T_p|$

best_distribution Character: name of the best-matching distribution

nb_diagnostic If test_nb=TRUE: list with T_NB and overdispersed (logical)

See Also

[auto_nmf_distribution](#), [nmf](#)

Examples

```
A <- abs(rsparsematrix(200, 100, 0.3))
model <- nmf(A, k = 5, loss = "mse")
diag <- score_test_distribution(A, model)
print(diag$scores)
cat("Best distribution:", diag$best_distribution, "\n")
```

simulateNMF

*Simulate an NMF dataset***Description**

Generate a random nonnegative matrix with known factor structure for benchmarking NMF recovery. Uses block-diagonal construction: each factor owns a disjoint subset of features (rows) and dominates a disjoint subset of samples (columns), with small cross-talk for realism. This produces clearly recoverable factors even at moderate noise levels. Inspired by `NMF::syntheticNMF`.

Usage

```
simulateNMF(nrow, ncol, k, noise = 0.5, dropout = 0, seed = NULL)
```

Arguments

nrow	number of rows (features)
ncol	number of columns (samples)
k	true rank (number of factors)
noise	noise level as a multiplier on the mean signal. A value of 1.0 means the noise standard deviation equals the mean signal value. Default: 0.5.
dropout	fraction of entries to set to zero (0 = no dropout). Default: 0.
seed	seed for random number generation

Value

list of dense matrix A and true w and h models

See Also

[simulateSwimmer](#), [nmf](#)

Examples

```
data <- simulateNMF(50, 30, k = 3, noise = 0.1, seed = 42)
dim(data$A) # 50 x 30
dim(data$w) # 50 x 3
dim(data$h) # 3 x 30
```

simulateSwimmer	<i>Simulate Swimmer Dataset</i>
-----------------	---------------------------------

Description

Generate a synthetic "swimmer" dataset consisting of stick figure images in various swimming poses. This is a classic benchmark dataset for NMF, originally described in Donoho and Stodden (2003). Each image is a 32x32 pixel representation of a stick figure with a torso and four limbs (left arm, right arm, left leg, right leg), where each limb can be in one of 4 positions.

Usage

```
simulateSwimmer(
  n_images = 256,
  style = c("stick", "gaussian"),
  sigma = 1.5,
  noise = 0,
  seed = NULL,
  return_factors = FALSE
)
```

Arguments

n_images	number of images to generate (default: 256 for all combinations)
style	either "stick" for stick figures or "gaussian" for smoothed Gaussian blobs (default: "stick")
sigma	standard deviation for Gaussian smoothing when style = "gaussian" (default: 1.5)
noise	standard deviation of Gaussian noise to add (default: 0)
seed	seed for random number generation
return_factors	logical, if TRUE return the true W and H factors (default: FALSE)

Details

The swimmer dataset consists of stick figure images with:

- A fixed torso (circle for head, vertical line for body)
- 4 limbs, each with 4 possible angles: 0, 45, 90, 135 degrees from body
- Total of 16 "limb factors" (4 limbs x 4 positions each)
- 256 possible combinations when sampling all positions

The true rank of this dataset is 16 (the number of unique limb positions). When `return_factors = TRUE`, the function returns the true generative factors W (256 x 16) and H (16 x 1024), where each column of H represents one limb position pattern.

Style options:

- style = "stick": Sharp lines for limbs (binary image)
- style = "gaussian": Smoothed with Gaussian kernel for softer edges

Value

If `return_factors = FALSE`, returns a sparse matrix (`n_images` x 1024) where each row is a flattened 32x32 image. If `return_factors = TRUE`, returns a list with components:

A The data matrix (`n_images` x 1024)

w True factor matrix *W* (`n_images` x 16)

h True factor matrix *H* (16 x 1024)

limb_positions Matrix (`n_images` x 4) indicating limb positions

References

Donoho, D. and Stodden, V. (2003). "When does non-negative matrix factorization give a correct decomposition into parts?" *Advances in Neural Information Processing Systems* 16.

See Also

[simulateNMF, nmf](#)

Examples

```
# Generate all 256 swimmer combinations
swimmers <- simulateSwimmer()
dim(swimmers) # 256 x 1024

# Generate random subset with Gaussian smoothing
swimmers_smooth <- simulateSwimmer(n_images = 100, style = "gaussian", seed = 123)

# Get true factors for validation
swimmer_data <- simulateSwimmer(return_factors = TRUE)
# Run NMF and compare to true factors
model <- nmf(swimmer_data$A, k = 16, maxit = 100)
```

sparsity

Compute the sparsity of each NMF factor

Description

Compute the sparsity of each NMF factor

Usage

```
sparsity(object, ...)

## S4 method for signature 'nmf'
sparsity(object, ...)
```

Arguments

object	object of class nmf.
...	additional parameters

Details

For `nmf` models, the sparsity of each factor is computed and summarized on w and h matrices. A long data.frame with columns factor, sparsity, and model is returned.

Value

A data.frame with columns factor, sparsity, and model ("w" or "h").

See Also

[nmf](#), [summary](#), [nmf-method](#)

Examples

```
data <- simulateNMF(50, 30, k = 3, seed = 1)
model <- nmf(data$A, 3, seed = 1, maxit = 50)
sparsity(model)
```

st_add_transpose	<i>Add Transpose Section to an Existing StreamPress File</i>
------------------	--

Description

Add Transpose Section to an Existing StreamPress File

Usage

```
st_add_transpose(path, verbose = TRUE)
```

Arguments

path	Path to a .spz file.
verbose	Logical; print progress. Default TRUE.

Value

Invisibly returns the path.

See Also

[st_write](#), [st_read](#)

Examples

```
library(Matrix)
m <- rsparsematrix(50, 20, 0.3)
tmp <- tempfile(fileext = ".spz")
st_write(m, tmp, include_transpose = FALSE)
st_add_transpose(tmp)
info <- st_info(tmp)
unlink(tmp)
```

st_chunk_ranges

Get Column Ranges for Each Chunk in a StreamPress File

Description

Returns the column ranges (1-indexed, inclusive) for each chunk without decompressing any data.

Usage

```
st_chunk_ranges(path)
```

Arguments

path Path to a .spz file.

Value

A data.frame with columns start and end.

See Also

[st_map_chunks](#)

Examples

```
## Not run:
ranges <- st_chunk_ranges("data.spz")
print(ranges)

## End(Not run)
```

st_filter_cols	<i>Slice Columns Matching Variable Metadata Filter</i>
----------------	--

Description

Slice Columns Matching Variable Metadata Filter

Usage

```
st_filter_cols(path, ..., threads = 0L)
```

Arguments

- path Path to a .spz file.
- ... Filter expression on var columns (passed to subset()).
- threads Integer decode threads. 0 = all.

Value

A dgCMatrix sparse matrix.

See Also

[st_filter_rows](#), [st_read_var](#)

Examples

```
## Not run:
mat <- st_filter_cols("data.spz", highly_variable == TRUE)
dim(mat)

## End(Not run)
```

st_filter_rows	<i>Slice Rows Matching Observation Metadata Filter</i>
----------------	--

Description

Slice Rows Matching Observation Metadata Filter

Usage

```
st_filter_rows(path, ..., threads = 0L)
```

Arguments

path	Path to a .spz file.
...	Filter expression on obs columns (passed to subset()).
threads	Integer decode threads. 0 = all.

Value

A dgCMatrx sparse matrix.

See Also

[st_filter_cols](#), [st_obs_indices](#)

Examples

```
## Not run:
mat <- st_filter_rows("data.spz", cell_type == "B cell")
dim(mat)

## End(Not run)
```

st_free_gpu

Free GPU-Resident Sparse Matrix

Description

Explicitly frees CUDA device memory held by a gpu_sparse_matrix object. This is optional — the memory will be freed automatically when the object is garbage-collected.

Usage

```
st_free_gpu(x)
```

Arguments

x	A gpu_sparse_matrix object from st_read_gpu().
---	--

Value

Invisibly returns NULL.

See Also

[st_read_gpu](#)

Examples

```
## Not run:  
gpu_mat <- st_read_gpu("matrix.spz")  
st_free_gpu(gpu_mat)  
  
## End(Not run)
```

st_info

Get metadata from a StreamPress file

Description

Reads only the header – no decompression is performed.

Usage

```
st_info(path)
```

Arguments

path Path to a .spz file.

Value

A list with fields including rows, cols, nnz, density_pct, file_bytes, raw_bytes, ratio, version, has_obs, has_var, has_transpose, transp_chunk_cols.

See Also

[st_read](#), [st_write](#)

Examples

```
library(Matrix)  
A <- rsparsematrix(100, 50, 0.1)  
f <- tempfile(fileext = ".spz")  
st_write(A, f)  
info <- st_info(f)  
cat(sprintf("Matrix: %d x %d, nnz=%d\n", info$rows, info$cols, info$nnz))  
unlink(f)
```

st_map_chunks	<i>Apply a Function to Every Chunk in a StreamPress File</i>
---------------	--

Description

Sequentially reads and decodes each chunk, applies fn, and collects results.

Usage

```
st_map_chunks(path, fn, transpose = FALSE, threads = 0L)
```

Arguments

path	Path to a .spz file.
fn	Function taking (chunk, col_start, col_end) where chunk is a dgCMatrix.
transpose	Logical; if TRUE, iterate over transpose chunks (row chunks of the original matrix). Default FALSE.
threads	Integer; decode threads per chunk. Default 0 (all threads).

Value

Invisible list of results from fn.

See Also

[st_chunk_ranges](#)

Examples

```
## Not run:
# Compute column sums per chunk
st_map_chunks("data.spz", function(chunk, s, e) Matrix::colSums(chunk))

## End(Not run)
```

st_obs_indices	<i>Get Row Indices Matching Observation Metadata Filter</i>
----------------	---

Description

Reads the obs table, applies a filter expression via `subset()`, and returns matching row indices.

Usage

```
st_obs_indices(path, ...)
```

Arguments

path	Path to a .spz file.
...	Filter expressions passed to <code>subset()</code> on the obs table.

Value

Integer vector of matching row indices (1-based).

See Also

[st_filter_rows](#), [st_read_obs](#)

Examples

```
## Not run:
idx <- st_obs_indices("data.spz", cell_type == "B cell")
length(idx)

## End(Not run)
```

st_read	<i>Read a StreamPress file into a dgCMatrix</i>
---------	---

Description

Read a StreamPress file into a dgCMatrix

Usage

```
st_read(path, cols = NULL, reorder = TRUE, threads = NULL)
```

Arguments

path	Path to a .spz file.
cols	Optional integer vector of column indices to read (1-indexed).
reorder	Logical; if TRUE (default), undo any row permutation.
threads	Integer or NULL; threads for parallel decompression. NULL (default) enables automatic selection: 1 thread for files smaller than 50 MB (where threading overhead exceeds the benefit), and all available threads for larger files. Use 0 to always request all available threads, or a positive integer to fix the count.

Value

A dgCMatrix sparse matrix with dimnames if available.

See Also

[st_write](#), [st_info](#)

Examples

```
library(Matrix)
A <- rsparsematrix(100, 50, 0.1)
f <- tempfile(fileext = ".spz")
st_write(A, f)
B <- st_read(f)
all.equal(A, B)
unlink(f)
```

st_read_dense	<i>Read a Dense Matrix from StreamPress v3 Format</i>
---------------	---

Description

Read a Dense Matrix from StreamPress v3 Format

Usage

```
st_read_dense(path)
```

Arguments

path	Path to a StreamPress v3 .spz file.
------	-------------------------------------

Value

A numeric matrix.

See Also[st_write_dense](#), [st_read](#)**Examples**

```
A <- matrix(rnorm(1000), 50, 20)
f <- tempfile(fileext = ".spz")
st_write_dense(A, f)
B <- st_read_dense(f)
max(abs(A - B))
unlink(f)
```

st_read_gpu

*Read StreamPress File Directly to GPU Memory***Description**

Reads a .spz v2 file and decodes it directly on the GPU, returning an opaque GPU-resident CSC matrix. This avoids the CPU-to-GPU transfer that occurs when using `st_read()` followed by `nmf(data, gpu = TRUE)`.

The returned object is an external reference — the matrix data lives entirely in GPU device memory. Pass it directly to `nmf()` for zero-copy GPU NMF.

Usage

```
st_read_gpu(path, device = 0L)
```

Arguments

path	Path to a .spz file (v2 format required).
device	Integer; CUDA device ID (default 0).

Details

Only .spz v2 format is supported for GPU decode. Use `st_convert()` to convert other formats to v2.

The returned object has a finalizer that automatically frees GPU memory when the R object is garbage-collected. You can also free it manually with `st_free_gpu()`.

Value

An object of class "gpu_sparse_matrix" with fields:

m Number of rows
n Number of columns
nnz Number of non-zeros
device CUDA device ID
.col_ptr Opaque device pointer (numeric)
.row_idx Opaque device pointer (numeric)
.values Opaque device pointer (numeric)

See Also

[st_read](#), [st_free_gpu](#), [nmf](#)

Examples

```
## Not run:
# Read directly to GPU
gpu_data <- st_read_gpu("data.spz")

# Run NMF on GPU-resident data (zero-copy)
result <- nmf(gpu_data, k = 10)

# Clean up (optional - GC will do this automatically)
st_free_gpu(gpu_data)

## End(Not run)
```

st_read_obs

Read Observation (Row) Metadata from a StreamPress File

Description

Reads the embedded obs table from a v2 .spz file without decompressing the matrix data. Returns an empty data.frame if no obs table was stored.

Usage

```
st_read_obs(path)
```

Arguments

path Path to a .spz file.

Value

A data.frame with observation metadata, or an empty data.frame if no obs table is present.

See Also

[st_read_var](#), [st_write](#)

Examples

```
## Not run:
obs <- st_read_obs("data.spz")
head(obs)

## End(Not run)
```

st_read_var

Read Variable (Column) Metadata from a StreamPress File

Description

Reads the embedded var table from a v2 .spz file without decompressing the matrix data. Returns an empty data.frame if no var table was stored.

Usage

```
st_read_var(path)
```

Arguments

path Path to a .spz file.

Value

A data.frame with variable metadata, or an empty data.frame if no var table is present.

See Also

[st_read_obs](#), [st_write](#)

Examples

```
## Not run:
var <- st_read_var("data.spz")
head(var)

## End(Not run)
```

st_slice

Slice Rows and/or Columns from a StreamPress File

Description

Convenience function combining row and column slicing.

Usage

```
st_slice(path, rows = NULL, cols = NULL, threads = 0L)
```

Arguments

path	Path to a .spz file.
rows	Optional integer vector of row indices (1-indexed).
cols	Optional integer vector of column indices (1-indexed).
threads	Integer; number of threads (0 = all available). Default 0.

Value

A dgCMatrix sparse matrix.

See Also

[st_slice_cols](#), [st_slice_rows](#)

Examples

```
## Not run:
mat <- st_slice("data.spz", rows = 1:100, cols = 1:10)
dim(mat)

## End(Not run)
```

st_slice_cols

Slice Columns from a StreamPress File

Description

Read a subset of columns from a .spz file without decompressing the entire matrix.

Usage

```
st_slice_cols(path, cols, threads = 0L)
```

Arguments

path	Path to a .spz file.
cols	Integer vector of column indices (1-indexed).
threads	Integer; number of threads (0 = all available). Default 0.

Value

A dgCMatrix sparse matrix.

See Also

[st_slice_rows](#), [st_slice](#)

Examples

```
## Not run:
mat <- st_slice_cols("data.spz", cols = 1:10)
dim(mat)

## End(Not run)
```

st_slice_rows

Slice Rows from a StreamPress File

Description

Read a subset of rows from a .spz file using the pre-stored transpose section. Requires that the file was written with include_transpose = TRUE.

Usage

```
st_slice_rows(path, rows, threads = 0L)
```

Arguments

path	Path to a .spz file.
rows	Integer vector of row indices (1-indexed).
threads	Integer; number of threads (0 = all available). Default 0.

Value

A dgCMatrix sparse matrix.

See Also

[st_slice_cols](#), [st_slice](#)

Examples

```
## Not run:
mat <- st_slice_rows("data.spz", rows = 1:100)
dim(mat)

## End(Not run)
```

st_write

Write a sparse matrix to a StreamPress file

Description

Write a sparse matrix to a StreamPress file

Usage

```
st_write(
  x,
  path,
  obs = NULL,
  var = NULL,
  delta = TRUE,
  value_pred = FALSE,
  verbose = FALSE,
  precision = "auto",
  row_sort = FALSE,
  include_transpose = TRUE,
  chunk_cols = NULL,
  chunk_bytes = 8e+06,
  transp_chunk_cols = NULL,
  transp_chunk_bytes = NULL,
  threads = 0L
)
```

Arguments

x	A sparse matrix (dgCMatrix) or object coercible to one.
path	Output file path. Extension .spz is recommended.
obs	Optional data.frame of observation (row/cell) metadata. Must have nrow(obs) == nrow(x).
var	Optional data.frame of variable (column/gene) metadata. Must have nrow(var) == ncol(x).
delta	Logical; use density-based delta prediction for structure. Default TRUE.
value_pred	Logical; use value prediction for integer-valued data. Default FALSE.

verbose	Logical; print compression statistics. Default FALSE.
precision	Value precision: "auto" (default), "fp32", "fp16", "quant8", "fp64".
row_sort	Logical; sort rows by nnz for better compression.
include_transpose	Logical; store $CSC(A^T)$ in the file. Default TRUE.
chunk_cols	Integer or NULL; columns per chunk. If NULL, computed from chunk_bytes.
chunk_bytes	Numeric; target bytes per chunk when chunk_cols is NULL. Default 8 MB, which yields ~50 columns per chunk for typical scRNA-seq matrices (~38 k rows). Smaller chunks create more parallel work during reading; larger chunks compress slightly better.
transp_chunk_cols	Integer or NULL; columns per transpose chunk.
transp_chunk_bytes	Numeric or NULL; target bytes per transpose chunk.
threads	Integer; number of threads for parallel compression (0 = all available). Default 0.

Value

Invisibly returns a list with compression statistics.

See Also

[st_read](#), [st_info](#)

Examples

```
library(Matrix)
A <- rsparsematrix(1000, 500, 0.05)
f <- tempfile(fileext = ".spz")
st_write(A, f)
B <- st_read(f)
all.equal(A, B) # TRUE
unlink(f)
```

st_write_dense

Write a Dense Matrix to StreamPress v3 Format

Description

Write a Dense Matrix to StreamPress v3 Format

Usage

```
st_write_dense(
  x,
  path,
  include_transpose = FALSE,
  chunk_cols = 2048L,
  codec = "raw",
  delta = FALSE,
  verbose = FALSE
)
```

Arguments

<code>x</code>	A numeric matrix.
<code>path</code>	Output file path. Extension <code>.spz</code> is recommended.
<code>include_transpose</code>	Logical; store transposed panels. Default <code>FALSE</code> .
<code>chunk_cols</code>	Integer; columns per chunk. Default 2048.
<code>codec</code>	Compression codec: <code>"raw"</code> , <code>"fp16"</code> , <code>"quant8"</code> , <code>"fp16_rans"</code> , <code>"fp32_rans"</code> . Default <code>"raw"</code> .
<code>delta</code>	Logical; apply XOR-delta encoding. Default <code>FALSE</code> .
<code>verbose</code>	Logical; print write statistics. Default <code>FALSE</code> .

Value

Invisibly returns a list with write statistics.

See Also

[st_read_dense](#), [st_write](#)

Examples

```
A <- matrix(rnorm(1000), 50, 20)
f <- tempfile(fileext = ".spz")
st_write_dense(A, f, include_transpose = TRUE)
B <- st_read_dense(f)
max(abs(A - B))
unlink(f)
```

st_write_list

Write a List of Matrices as a Single StreamPress File

Description

Column-concatenates a list of matrices and writes them as a single .spz file. All matrices must have the same number of rows.

Usage

```
st_write_list(
  x,
  path,
  obs = NULL,
  var = NULL,
  chunk_bytes = 6.4e+07,
  chunk_cols = NULL,
  include_transpose = TRUE,
  precision = "auto",
  threads = 0L,
  verbose = FALSE
)
```

Arguments

x	A list of dgCMatrix objects (or coercible). All must have identical nrow.
path	Output .spz path.
obs	Optional data.frame of cell metadata (nrow == total cols).
var	Optional data.frame of gene metadata (nrow == nrow of mats).
chunk_bytes	Target bytes per chunk. Default 64 MB.
chunk_cols	Explicit column count per chunk. Overrides chunk_bytes.
include_transpose	Logical. Default TRUE.
precision	Value precision. Default "auto".
threads	Integer. 0 = all threads.
verbose	Logical.

Value

Invisibly, compression statistics.

See Also

[st_write](#)

Examples

```
## Not run:
mats <- list(mat1, mat2)
st_write_list(mats, "combined.spz")

## End(Not run)
```

streampress	<i>StreamPress I/O: Read, Write, and Inspect Compressed Matrices</i>
-------------	--

Description

Functions for reading and writing matrices in StreamPress (.spz) format. StreamPress achieves 5-10x compression over raw float32 CSC binary on typical scRNA-seq sparse matrices using rANS entropy coding. Beyond storage savings, SPZ is also faster to read than raw CSC binary at any thread count: the bottleneck in reading large sparse matrices is sparse object construction (sorting indices, allocating R/Eigen structures), which SPZ parallelises across independent chunks while raw CSC must perform sequentially.

Details

StreamPress (.spz) supports two format versions:

- **v2 (sparse)**: Column-oriented compressed CSC format. Lossless, 5-10x compression over raw float32 CSC. Self-describing header.
- **v3 (dense)**: Column-major dense panels with optional FP16/QUANT8/rANS compression. For streaming NMF on dense data.

See Also

```
st_write, st_read, st_info, st_write_dense, st_read_dense
```

subset,nmf-method	<i>nmf class methods</i>
-------------------	--------------------------

Description

Given an NMF model in the form $A = wdh$, project projects w onto A to solve for h .

Usage

```
## S4 method for signature 'nmf'
subset(x, i, ...)

## S4 method for signature 'nmf,ANY,ANY,ANY'
x[i]

## S4 method for signature 'nmf'
head(x, n = getOption("digits"), ...)

## S4 method for signature 'nmf'
show(object)

## S4 method for signature 'nmf'
dimnames(x)

## S4 method for signature 'nmf'
dim(x)

## S4 method for signature 'nmf'
t(x)

## S4 method for signature 'nmf'
sort(x, decreasing = TRUE, ...)

## S4 method for signature 'nmf'
prod(x, ..., na.rm = FALSE)

## S4 method for signature 'nmf'
x$name

## S4 method for signature 'nmf,list'
coerce(from, to)

## S4 method for signature 'nmf'
x[[i]]

## S4 method for signature 'nmf'
predict(
  object,
  data,
  L1 = NULL,
  L2 = NULL,
  mask = NULL,
  upper_bound = NULL,
  threads = 0,
  verbose = FALSE,
  ...)
```

)

Arguments

x	object of class nmf.
i	indices
...	arguments passed to or from other methods
n	number of rows/columns to show
object	fitted model, class nmf, generally the result of calling nmf, with models of equal dimensions as data
decreasing	logical. Should the sort be increasing or decreasing?
na.rm	remove na values
name	name of nmf class slot
from	class which the coerce method should perform coercion from
to	class which the coerce method should perform coercion to
data	dense or sparse matrix of features in rows and samples in columns. Prefer matrix or Matrix::dgCMatrix, respectively. Also accepts a file path (character string) which will be auto-loaded based on extension.
L1	a single LASSO penalty in the range (0, 1]
L2	a single Ridge penalty greater than zero
mask	missing data mask. Accepts: NULL (no masking), "zeros" (mask zeros), "NA" (mask NAs), a dgCMatrix/matrix (custom mask), or list("zeros", <matrix>) to mask zeros and use a custom mask simultaneously.
upper_bound	maximum value permitted in least squares solutions, essentially a bounded-variable least squares problem between 0 and upper_bound
threads	number of threads for OpenMP parallelization (default 0 = all available)
verbose	print progress information (default FALSE)

Details

For the alternating least squares matrix factorization update problem $A = wh$, the updates (or projection) of h is given by the equation:

$$w^T wh = wA_j$$

which is in the form $ax = b$ where $a = w^T w$, $x = h$ and $b = wA_j$ for all columns j in A .

Any L1 penalty is subtracted from b and should generally be scaled to $\max(b)$, where $b = wA_j$ for all columns j in A . An easy way to properly scale an L1 penalty is to normalize all columns in w to sum to the same value (e.g. 1). No scaling is applied in this function. Such scaling guarantees that $L1 = 1$ gives a completely sparse solution.

There are specializations for dense and sparse input matrices, symmetric input matrices, and for rank-1 and rank-2 projections. See documentation for [nmf](#) for theoretical details and guidance.

Value

An nmf object subsetting to the specified factors.
 An nmf object subsetting to the specified factors.
 Invisibly returns the nmf object.
 Invisibly returns the nmf object.
 A list of length two: row names of w and column names of h.
 An integer vector of length 3: number of features, number of samples, and rank.
 An nmf object with transposed factorization (w and h swapped and transposed).
 An nmf object with factors reordered by decreasing (or increasing) d.
 A dense matrix equal to $w \%*\% \text{diag}(d) \%*\% h$.
 Contents of the named slot (w, d, h, or misc), or the named element from misc.
 A named list with elements w, d, h, and misc.
 The element from the misc list at the given name or index.
 An [nmf](#) object containing the projected model (with updated h matrix).

Author(s)

Zach DeBruine

References

DeBruine, ZJ, Melcher, K, and Triche, TJ. (2021). "High-performance non-negative matrix factorization for large single-cell data." *BioRxiv*.

See Also

[nnls](#), [nmf](#)

Examples

```
library(Matrix)
w <- matrix(runif(1000 * 10), 1000, 10)
h_true <- matrix(runif(10 * 100), 10, 100)
# A is the crossproduct of "w" and "h" with 10% signal dropout
mask <- rsparsematrix(1000, 100, density = 0.1)
A <- (w %*% h_true) * (mask != 0)
h <- nnls(w = w, A = A)
cor(as.vector(h_true), as.vector(h))

# alternating projections refine solution (like NMF)
h <- nnls(w = w, A = A)
w <- nnls(h = h, A = A)
h <- nnls(w = w, A = A)
w <- nnls(h = h, A = A)
h <- nnls(w = w, A = A)
w <- nnls(h = h, A = A)
```

summary,nmf-method	<i>Summarize NMF factors</i>
--------------------	------------------------------

Description

summary method for class "nmf". Describes metadata representation in NMF factors. Returns object of class nmfSummary. Plot results using plot.

Usage

```
## S4 method for signature 'nmf'
summary(object, group_by, stat = "sum", ...)

## S3 method for class 'nmfSummary'
plot(x, ...)
```

Arguments

object	an object of class "nmf", usually, a result of a call to nmf
group_by	a discrete factor giving groupings for samples or features. Must be of the same length as number of samples in object\$h or number of features in object\$w.
stat	either sum (sum of factor weights falling within each group), or mean (mean factor weight falling within each group).
...	Additional arguments (unused).
x	nmfSummary object, the result of calling summary on an nmf object

Value

data.frame with columns group, factor, and stat
 A ggplot2 object showing factor representation by group.

See Also

[nmf](#), [sparsity](#)
[summary,nmf-method](#), [nmf](#)

Examples

```
library(Matrix)
A <- rsparsematrix(100, 50, 0.1)
model <- nmf(A, k = 3, seed = 42, tol = 1e-2, maxit = 10)
groups <- factor(sample(c("A", "B"), 50, replace = TRUE))
s <- summary(model, group_by = groups)
```

```
library(Matrix)
A <- rsparsematrix(100, 50, 0.1)
model <- nmf(A, k = 3, seed = 42, tol = 1e-2, maxit = 10)
groups <- factor(sample(c("A", "B"), 50, replace = TRUE))
s <- summary(model, group_by = groups)
if (requireNamespace("ggplot2", quietly = TRUE)) {
  plot(s)
}
```

summary.consensus_nmf *Summary for Consensus NMF*

Description

Summary for Consensus NMF

Usage

```
## S3 method for class 'consensus_nmf'
summary(object, ...)
```

Arguments

object	consensus_nmf object
...	additional arguments (unused)

Value

Invisibly returns the consensus_nmf object. Summary statistics are printed to the console.

See Also

[consensus_nmf](#), [plot.consensus_nmf](#)

Examples

```
library(Matrix)
A <- rsparsematrix(50, 30, 0.3)
cons <- consensus_nmf(A, k = 3, reps = 5, seed = 42)
summary(cons)
```

```
summary.factor_net_result
```

Summarize a factor_net_result

Description

Summarize a factor_net_result

Usage

```
## S3 method for class 'factor_net_result'  
summary(object, ...)
```

Arguments

object	A factor_net_result object.
...	Additional arguments (unused).

Value

Invisibly returns object.

See Also

[factor_net](#), [fit](#), [print.factor_net_result](#)

```
summary.fn_classifier_eval
```

Summarize a classifier evaluation result

Description

Returns a tidy data frame of aggregate classification metrics.

Usage

```
## S3 method for class 'fn_classifier_eval'  
summary(object, ...)
```

Arguments

object	An fn_classifier_eval object returned by classify_embedding , classify_logistic , or classify_rf .
...	Additional arguments (unused).

Value

A data frame with columns `metric` and `value`.

See Also

[print.fn_classifier_eval](#), [classify_embedding](#)

svd	<i>Truncated SVD / PCA with constraints and regularization</i>
-----	--

Description

Compute a truncated SVD or PCA of a matrix using multiple algorithm backends. Supports sparse and dense matrices, implicit PCA centering, per-element regularization (L1/L2/non-negativity/bounds), Gram-level regularization (L2,1/angular/graph Laplacian), and automatic rank selection via speckled holdout cross-validation.

Usage

```
svd(
  A,
  k = 10,
  tol = 1e-05,
  maxit = 200,
  center = FALSE,
  scale = FALSE,
  verbose = FALSE,
  seed = NULL,
  L1 = 0,
  L2 = 0,
  nonneg = FALSE,
  upper_bound = 0,
  test_fraction = 0,
  mask = NULL,
  robust = FALSE,
  resource = "auto",
  method = "auto",
  ...
)
```

Arguments

- | | |
|---|---|
| A | Input matrix. May be dense (<code>matrix</code>), sparse (<code>dgCMatrix</code>), or a path to a <code>.spz</code> file for out-of-core streaming SVD. |
| k | Number of factors (rank). Use "auto" for automatic rank selection via cross-validation. Default: 10. |

tol	Convergence tolerance per rank-1 subproblem. Default: 1e-5.
maxit	Maximum ALS iterations per factor. Default: 200.
center	If TRUE, subtract row means (PCA mode). Default: FALSE.
scale	If TRUE, divide each row by its standard deviation after centering. Default: FALSE.
verbose	Print per-factor diagnostics. Default: FALSE.
seed	Random seed for initialization and cross-validation. Default: NULL.
L1	L1 (lasso) penalty. Single value or length-2 vector. Default: 0.
L2	L2 (ridge) penalty. Single value or length-2 vector. Default: 0.
nonneg	Non-negativity constraints. Single logical or length-2 vector. Default: FALSE.
upper_bound	Upper bound constraints. Single value or length-2 vector. Default: 0.
test_fraction	Fraction of entries to hold out for cross-validation. Default: 0.
mask	Masking control. NULL (default, no masking), "zeros" (mask zero entries — only non-zero entries can be holdout in CV), a dgCMatix (custom mask matrix where non-zero entries are excluded), or list("zeros", <dgCMatix>) to combine both.
robust	Robustness control: FALSE (default), TRUE (Huber delta=1.345), "mae" (near-MAE), or a positive numeric Huber delta.
resource	Compute backend: "auto" (default), "cpu", or "gpu".
method	Algorithm: "auto" (default), "deflation", "krylov", "lanczos", "irlba", "randomized".
...	Advanced parameters. See Advanced Parameters section.

Details

Multiple algorithms are available via the method parameter:

"deflation" Sequential rank-1 ALS with deflation correction. Supports all constraints/CV/auto-rank. Best for small k with mixed constraints.

"krylov" Krylov-Seeded Projected Refinement (KSPR). Block method: computes all k factors simultaneously via Lanczos seed + Gram-solve-then-project. Faster than deflation for larger k. Supports all regularization except robust. CV is evaluated post-convergence.

"lanczos" Unconstrained Lanczos bidiagonalization. Fast for unconstrained SVD. No regularization support.

"irlba" Implicitly Restarted Lanczos. Good general-purpose unconstrained SVD. No regularization support.

"randomized" Randomized SVD with power iterations. Fast approximate SVD for large matrices. No regularization support.

Value

An S4 object of class `svd` with slots:

`u` Left singular vectors (scores), $m \times k$ matrix
`d` Singular values, length- k numeric vector
`v` Right singular vectors (loadings), $n \times k$ matrix
`misc` List with metadata: `centered`, `row_means`, `test_loss`, `iters_per_factor`, `wall_time_ms`

Auto-rank

When `k = "auto"`, a speckled holdout set is created and test MSE is evaluated after each factor. Rank selection stops when test MSE fails to improve for patience consecutive factors. The returned model is truncated to the best rank.

Advanced Parameters (via ...)

The following parameters can be passed via `...`:

`L21` L2,1 (group sparsity) penalty. Single value or length-2 vector (default 0).
`angular` Angular (orthogonality) penalty. Single value or length-2 vector (default 0).
`graph_U` Sparse graph Laplacian for features ($m \times m$). Default NULL.
`graph_V` Sparse graph Laplacian for samples ($n \times n$). Default NULL.
`graph_lambda` Graph regularization strength. Single value or length-2 (default 0).
`convergence` Convergence criterion: "factor" (default) or "global".
`cv_seed` Separate seed for holdout mask. Default NULL.
`patience` Auto-rank non-improving factor patience (default 3).
`k_max` Maximum rank for auto-rank mode (default 50).
`threads` Number of OpenMP threads. 0 = all (default 0).

Note

This function shadows `base::svd()`. To use the base R version, call `base::svd()` explicitly.

See Also

[nmf](#), [pca](#)

Examples

```
library(RcppML)
data(iris)
A <- as.matrix(iris[, 1:4])

# Standard SVD (3 factors)
s <- svd(A, k = 3)

# PCA with centering (convenience wrapper)
```

```

p <- pca(A, k = 3)

# Auto-rank PCA
ar <- svd(A, k = "auto", center = TRUE, verbose = TRUE)

# Robust PCA (outlier-resistant)
rp <- svd(A, k = 3, center = TRUE, robust = TRUE, verbose = TRUE)

```

svd-class

svd S4 Class

Description

S4 class for deflation SVD/PCA results.

Computes scores for new samples by projecting newdata onto the right singular vectors. Equivalent to PCA "out-of-sample prediction".

Usage

```

## S4 method for signature 'svd,ANY,ANY,ANY'
x[i]

## S4 method for signature 'svd'
head(x, n = getOption("digits"), ...)

## S4 method for signature 'svd'
show(object)

## S4 method for signature 'svd'
dim(x)

reconstruct(object, ...)

## S4 method for signature 'svd'
reconstruct(object, ...)

## S4 method for signature 'svd'
predict(object, newdata, ...)

variance_explained(object, ...)

## S4 method for signature 'svd'
variance_explained(object, ...)

```

Arguments

<code>x</code>	object of class <code>svd</code>
<code>i</code>	indices for subsetting factors
<code>n</code>	number of rows/columns to show
<code>...</code>	Ignored.
<code>object</code>	An <code>svd</code> object
<code>newdata</code>	A numeric matrix of new samples (rows = samples, columns = features). Must have the same number of features as the original data (<code>ncol(newdata) == nrow(object@v)</code>).

Value

A subsetting `svd` object containing only the selected factors.

Invisibly returns the `svd` object `x`.

Invisibly returns the `svd` object.

Integer vector of length 3: `c(m, n, k)` where `m` is the number of rows, `n` the number of columns, and `k` the rank.

Dense matrix: $U \cdot \text{diag}(d) \cdot V'$ (plus row means if centered)

A numeric matrix of scores with `nrow(newdata)` rows and `length(object@d)` columns (i.e., same k as the model). Each row is the projection of one new sample onto the singular space.

Numeric vector of proportion of variance explained by each factor

Slots

`u` left singular vectors (scores), `m x k` matrix

`d` singular values, numeric vector of length `k`

`v` right singular vectors (loadings), `n x k` matrix

`misc` list containing metadata: `centered`, `row_means`, `test_loss`, `iters_per_factor`, `wall_time_ms`, `auto_rank`

See Also

[svd](#), [pca](#), [reconstruct](#), [variance_explained](#)

Examples

```
A <- matrix(rnorm(200), 20, 10)
s <- RcppML::svd(A, k = 3)
Ahat <- reconstruct(s)
dim(Ahat)
```

```
A <- matrix(rnorm(200), 20, 10)
s <- RcppML::svd(A, k = 3)
variance_explained(s)
```

svd_layer

*Create an SVD/PCA factorization layer***Description**

Creates an unconstrained (SVD/PCA) factorization layer. No non-negativity constraint by default; factors are signed.

Usage

```
svd_layer(
  input,
  k,
  L1 = 0,
  L2 = 0,
  L21 = 0,
  angular = 0,
  upper_bound = 0,
  center = FALSE,
  scale = FALSE,
  method = c("auto", "deflation", "krylov", "lanczos", "irlba", "randomized"),
  mask = NULL,
  robust = FALSE,
  W = NULL,
  H = NULL,
  name = NULL,
  ...
)
```

Arguments

input	An <code>fn_node</code> (input, shared, condition, or another layer).
k	Factorization rank.
L1	L1 penalty (shared by U and V unless overridden). Default 0.
L2	L2 penalty. Default 0.
L21	Group sparsity penalty. Default 0.
angular	Orthogonality penalty. Default 0.
upper_bound	Box constraint. Default 0.
center	Center columns before factorization (PCA mode). Default FALSE.
scale	Scale columns to unit variance. Default FALSE.
method	SVD algorithm: "auto", "deflation", "krylov", "lanczos", "irlba", or "randomized". Default "auto".
mask	Masking mode: NULL (none), "zeros", or a sparse mask matrix. See svd for details.

robust	Use robust loss. Default FALSE.
W	Optional W() config to override U-specific settings.
H	Optional H() config to override V-specific settings.
name	Optional layer name (for results access).
...	Additional arguments forwarded to svd at fit time. Supports: convergence, cv_seed, patience, k_max, graph_U, graph_V, graph_lambda, threads. See ?svd for the complete list.

Details

SVD-specific parameters (center, scale, method) control the SVD algorithm directly. Regularization parameters and ... are forwarded to [svd](#) at fit time.

Value

An fn_node of type "svd_layer".

See Also

[nmf_layer](#), [factor_input](#), [factor_net](#), [svd](#)

Examples

```
data(aml)
inp <- factor_input(aml)
layer <- svd_layer(inp, k = 5)

# PCA with centering and scaling
pca_layer <- svd_layer(inp, k = 10, center = TRUE, scale = TRUE)
```

training_logger

Create a training logger for factor network fitting

Description

Returns a logger object that records per-iteration metrics during `fit()`. After training, the log can be printed, plotted, or exported to CSV.

Usage

```
training_logger(
  log_loss = TRUE,
  log_norms = FALSE,
  log_classifier = NULL,
  interval = 1L
)
```

Arguments

<code>log_loss</code>	Log reconstruction loss per layer (default TRUE).
<code>log_norms</code>	Log factor W/H norms per layer (default FALSE).
<code>log_classifier</code>	Evaluate classifier accuracy per iteration using a supplied <code>classify_embedding</code> configuration. Must be a list with <code>labels</code> and optionally <code>test_idx</code> , <code>k</code> , <code>side</code> , and <code>layer</code> (default NULL = disabled).
<code>interval</code>	Log every <code>interval</code> iterations (default 1).

Value

A `training_logger` object to pass to `fit()`.

See Also

[nmf](#), [fit](#)

Examples

```
## Not run:
logger <- training_logger(
  log_norms = TRUE,
  log_classifier = list(labels = labels, test_idx = test_idx, k = 5)
)
res <- fit(net, logger = logger)
print(logger)
plot(logger)
export_log(logger, "training_log.csv")

## End(Not run)
```

W

Per-factor configuration for factorization layers

Description

Use `W()` and `H()` inside `nmf_layer()` or `svd_layer()` to set factor-specific regularization, constraints, and targets. Parameters not set inherit from the layer defaults.

Usage

```
W(
  L1 = NULL,
  L2 = NULL,
  L21 = NULL,
  angular = NULL,
  upper_bound = NULL,
  nonneg = NULL,
```

```

graph = NULL,
graph_lambda = NULL,
target = NULL,
target_lambda = NULL
)

H(
  L1 = NULL,
  L2 = NULL,
  L21 = NULL,
  angular = NULL,
  upper_bound = NULL,
  nonneg = NULL,
  graph = NULL,
  graph_lambda = NULL,
  target = NULL,
  target_lambda = NULL
)

```

Arguments

L1	L1 (lasso) penalty. Default 0.
L2	L2 (ridge) penalty. Default 0.
L21	Group sparsity penalty. Default 0.
angular	Orthogonality penalty. Default 0.
upper_bound	Box constraint upper bound (0 = none). Default 0.
nonneg	Non-negativity constraint. Default TRUE for NMF, FALSE for SVD.
graph	Sparse graph Laplacian matrix (or NULL).
graph_lambda	Graph regularization strength. Default 0.
target	Target matrix for regularization (k x n for H, k x m for W). See compute_target for constructing targets from labels.
target_lambda	Target regularization strength. Default 0.

Value

An `fn_factor_config` object.

See Also

[factor_config](#), [nmf_layer](#), [factor_net](#)

Examples

```

# Per-factor config for W with L1 sparsity
w_cfg <- W(L1 = 0.1, nonneg = TRUE)
h_cfg <- H(L2 = 0.01)

```

Index

- * **datasets**
 - aml, [7](#)
 - digits, [31](#)
 - golub, [42](#)
 - hawaiiibirds, [45](#)
 - movielens, [47](#)
 - olivetti, [58](#)
 - pbmc3k, [59](#)
- [, nmf, ANY, ANY, ANY-method
 - (subset, nmf-method), [96](#)
- [, svd, ANY, ANY, ANY-method (svd-class), [106](#)
- [[, nmf-method (subset, nmf-method), [96](#)
- \$.nmf-method (subset, nmf-method), [96](#)
- \$.factor_net_result, [6](#)

- align, [4](#), [6](#), [13](#), [53](#)
- align, nmf-method (align), [6](#)
- aml, [7](#)
- as.data.frame.nmf_assessment, [8](#)
- as.data.frame.training_logger, [9](#), [34](#)
- assess, [9](#)
- auto_nmf_distribution, [4](#), [11](#), [30](#), [31](#), [75](#)

- bipartiteMatch, [5](#), [7](#), [13](#), [25](#)
- bipartition, [5](#), [14](#), [28](#), [29](#)
- biplot, nmf-method, [16](#)

- classify_embedding, [17](#), [19](#), [20](#), [69](#), [102](#), [103](#)
- classify_logistic, [18](#), [18](#), [20](#), [69](#), [102](#)
- classify_rf, [18](#), [19](#), [19](#), [69](#), [102](#)
- coerce, nmf, list-method
 - (subset, nmf-method), [96](#)
- compare_nmf, [20](#), [64](#)
- compute_target, [21](#), [51](#), [73](#), [111](#)
- consensus_nmf, [4](#), [13](#), [22](#), [62](#), [101](#)
- cosine, [5](#), [7](#), [24](#)
- cross_validate_graph, [5](#), [25](#), [39](#), [41](#), [68](#)

- dclust, [5](#), [15](#), [27](#), [63](#)

- diagnose_dispersion, [29](#)
- diagnose_zero_inflation, [12](#), [30](#)
- digits, [31](#)
- dim, nmf-method (subset, nmf-method), [96](#)
- dim, svd-method (svd-class), [106](#)
- dim.gpu_sparse_matrix
 - (gpu_sparse_matrix-methods), [44](#)
- dimnames, nmf-method
 - (subset, nmf-method), [96](#)

- evaluate, [4](#), [32](#), [53](#)
- evaluate, nmf-method (evaluate), [32](#)
- export_log, [34](#)

- factor_add, [5](#), [34](#), [35](#), [40](#)
- factor_concat, [5](#), [35](#), [35](#), [40](#)
- factor_condition, [36](#)
- factor_config, [5](#), [26](#), [36](#), [39](#), [70](#), [111](#)
- factor_input, [5](#), [38](#), [55](#), [71](#), [109](#)
- factor_net, [5](#), [6](#), [26](#), [35](#), [36](#), [38](#), [39](#), [39–41](#), [55](#), [67–71](#), [102](#), [109](#), [111](#)
- factor_shared, [5](#), [35](#), [40](#)
- fit, [5](#), [6](#), [26](#), [38](#), [39](#), [41](#), [67](#), [69](#), [102](#), [110](#)

- golub, [42](#)
- gpu_available, [5](#), [43](#), [44](#)
- gpu_info, [5](#), [43](#), [44](#)
- gpu_sparse_matrix-methods, [44](#)

- H, [5](#), [70](#)
- H(W), [110](#)
- hawaiiibirds, [45](#)
- head, nmf-method (subset, nmf-method), [96](#)
- head, svd-method (svd-class), [106](#)

- movielens, [47](#)
- mse, [53](#)

- ncol.gpu_sparse_matrix
 - (gpu_sparse_matrix-methods), [44](#)

- nmf, [4](#), [7](#), [9](#), [12](#), [14–16](#), [20–23](#), [25](#), [26](#), [28–31](#),
[33](#), [37](#), [43](#), [44](#), [48](#), [53–55](#), [57](#), [64](#), [65](#),
[73](#), [75](#), [76](#), [78](#), [79](#), [88](#), [98–100](#), [105](#),
[110](#)
- nmf-class, [53](#)
- nmf_layer, [5](#), [36](#), [38](#), [39](#), [54](#), [71](#), [109](#), [111](#)
- npls, [5](#), [53](#), [55](#), [99](#)
- nrow.gpu_sparse_matrix
(gpu_sparse_matrix-methods), [44](#)
- olivetti, [58](#)
- pbmc3k, [59](#)
- pca, [4](#), [61](#), [105](#), [107](#)
- plot.consensus_nmf, [23](#), [61](#), [101](#)
- plot.dclust, [63](#)
- plot.dendrogram, [63](#)
- plot.nmf, [20](#), [64](#)
- plot.nmfCrossValidate, [65](#)
- plot.nmfSummary (summary, nmf-method),
[100](#)
- plot.training_logger, [9](#), [66](#)
- predict, [53](#)
- predict, nmf-method, [4](#)
- predict, nmf-method (subset, nmf-method),
[96](#)
- predict, svd-method (svd-class), [106](#)
- predict.factor_net_result, [41](#), [66](#)
- print.factor_net, [67](#)
- print.factor_net_cv, [68](#)
- print.factor_net_result, [68](#), [102](#)
- print.fn_classifier_eval, [69](#), [103](#)
- print.fn_factor_config, [69](#)
- print.fn_global_config, [70](#)
- print.fn_node, [71](#)
- print.gpu_sparse_matrix
(gpu_sparse_matrix-methods), [44](#)
- print.nmf_assessment, [71](#)
- print.training_logger, [72](#)
- prod, nmf-method (subset, nmf-method), [96](#)
- RcppML (RcppML-package), [4](#)
- RcppML-package, [4](#)
- reconstruct, [4](#), [107](#)
- reconstruct (svd-class), [106](#)
- reconstruct, svd-method (svd-class), [106](#)
- refine, [22](#), [72](#)
- score_test_distribution, [12](#), [74](#)
- show, nmf-method (subset, nmf-method), [96](#)
- show, svd-method (svd-class), [106](#)
- simulateNMF, [4](#), [76](#), [78](#)
- simulateSwimmer, [76](#), [77](#)
- sort, nmf-method (subset, nmf-method), [96](#)
- sparsity, [5](#), [78](#), [100](#)
- sparsity, nmf-method (sparsity), [78](#)
- st_add_transpose, [79](#)
- st_chunk_ranges, [80](#), [84](#)
- st_filter_cols, [81](#), [82](#)
- st_filter_rows, [81](#), [81](#), [85](#)
- st_free_gpu, [5](#), [45](#), [82](#), [88](#)
- st_info, [5](#), [83](#), [86](#), [93](#), [96](#)
- st_map_chunks, [80](#), [84](#)
- st_obs_indices, [82](#), [85](#)
- st_read, [5](#), [60](#), [79](#), [83](#), [85](#), [87](#), [88](#), [93](#), [96](#)
- st_read_dense, [86](#), [94](#), [96](#)
- st_read_gpu, [5](#), [44](#), [45](#), [82](#), [87](#)
- st_read_obs, [5](#), [85](#), [88](#), [89](#)
- st_read_var, [5](#), [81](#), [89](#), [89](#)
- st_slice, [90](#), [91](#)
- st_slice_cols, [90](#), [90](#), [91](#)
- st_slice_rows, [90](#), [91](#), [91](#)
- st_write, [5](#), [79](#), [83](#), [86](#), [89](#), [92](#), [94–96](#)
- st_write_dense, [87](#), [93](#), [96](#)
- st_write_list, [95](#)
- streampress, [96](#)
- subset, nmf-method, [96](#)
- summary, nmf-method, [100](#)
- summary.consensus_nmf, [23](#), [62](#), [101](#)
- summary.factor_net_result, [69](#), [102](#)
- summary.fn_classifier_eval, [69](#), [102](#)
- svd, [4](#), [26](#), [37](#), [61](#), [103](#), [107–109](#)
- svd-class, [106](#)
- svd_layer, [5](#), [39](#), [55](#), [108](#)
- t, nmf-method (subset, nmf-method), [96](#)
- training_logger, [34](#), [66](#), [72](#), [109](#)
- variance_explained, [4](#), [107](#)
- variance_explained (svd-class), [106](#)
- variance_explained, svd-method
(svd-class), [106](#)
- W, [5](#), [70](#), [110](#)