Package ‘rsparse’

February 17, 2025

Type Package

Title Statistical Learning on Sparse Matrices

Version 0.5.3

Maintainer Dmitriy Selivanov <selivanov.dmitriy@gmail.com>

Description Implements many algorithms for statistical learning on
sparse matrices - matrix factorizations, matrix completion,
elastic net regressions, factorization machines.
Also 'rsparse' enhances 'Matrix' package by providing methods for
multithreaded <sparse, dense> matrix products and native slicing of
the sparse matrices in Compressed Sparse Row (CSR) format.
List of the algorithms for regression problems:
1) Elastic Net regression via Follow The Proximally-Regularized Leader (FTRL)
Stochastic Gradient Descent (SGD), as per McMahan et al(, <doi:10.1145/2487575.2488200>)
2) Factorization Machines via SGD, as per Rendle (2010, <doi:10.1109/ICDM.2010.127>)
List of algorithms for matrix factorization and matrix completion:
1) Weighted Regularized Matrix Factorization (WRMF) via Alternating Least
Squares (ALS) - paper by Hu, Koren, Volinsky (2008, <doi:10.1109/ICDM.2008.22>)
2) Maximum-Margin Matrix Factorization via ALS, paper by Rennie, Srebro
(2005, <doi:10.1145/1102351.1102441>)
3) Fast Truncated Singular Value Decomposition (SVD), Soft-Thresholded SVD,
Soft-Impute matrix completion via ALS - paper by Hastie, Mazumder
et al. (2014, <doi:10.48550/arXiv.1410.2596>)
4) Linear-Flow matrix factorization, from 'Practical linear models for
large-scale one-class collaborative filtering' by Sedhain, Bui, Kawale et al
(2016, ISBN:978-1-57735-770-4)
5) GlobalVectors (GloVe) matrix factorization via SGD, paper by Pennington,
Socher, Manning (2014, <https://aclanthology.org/D14-1162/>)
Package is reasonably fast and memory efficient - it allows to work with large
datasets - millions of rows and millions of columns. This is particularly useful
for practitioners working on recommender systems.

License GPL (>= 2)

Encoding UTF-8

LazyData true

ByteCompile true

Depends R (>= 3.6.0), methods, Matrix (>= 1.3)

Imports MatrixExtra (>= 0.1.7), Rcpp (>= 0.11), data.table (>= 1.10.0), float (>= 0.2-2), RhpcBLASctl, lgr (>= 0.2)

LinkingTo Rcpp, RcppArmadillo (>= 0.9.100.5.0)

Suggests testthat, covr

StagedInstall TRUE

URL `https://github.com/dselivanov/rsparse`

BugReports `https://github.com/dselivanov/rsparse/issues`

RoxygenNote 7.3.1

NeedsCompilation yes

Author Dmitriy Selivanov [aut, cre, cph] (<https://orcid.org/0000-0001-5413-1506>),
David Cortes [ctb],
Drew Schmidt [ctb] (configure script for BLAS, LAPACK detection),
Wei-Chen Chen [ctb] (configure script and work on linking to float package)

Repository CRAN

Date/Publication 2025-02-17 00:10:02 UTC

Contents

detect_number_omp_threads 2
FactorizationMachine 3
FTRL 5
GloVe 7
LinearFlow 9
metrics 11
movielens100k 12
PureSVD 13
ScaleNormalize 15
soft_impute 16
WRMF 18

Index **21**

`detect_number_omp_threads` *Detects number of OpenMP threads in the system*

Description

Detects number of OpenMP threads in the system respecting environment variables such as `OMP_NUM_THREADS` and `OMP_THREAD_LIMIT`

Usage

```
detect_number_omp_threads()
```

`FactorizationMachine` *Second order Factorization Machines*

Description

Creates second order Factorization Machines model

Methods

Public methods:

- `FactorizationMachine$new()`
- `FactorizationMachine$partial_fit()`
- `FactorizationMachine$fit()`
- `FactorizationMachine$predict()`
- `FactorizationMachine$clone()`

Method `new()`**:** creates Creates second order Factorization Machines model

Usage:

```
FactorizationMachine$new(
  learning_rate_w = 0.2,
  rank = 4,
  lambda_w = 0,
  lambda_v = 0,
  family = c("binomial", "gaussian"),
  intercept = TRUE,
  learning_rate_v = learning_rate_w
)
```

Arguments:

`learning_rate_w` learning rate for features intercations

`rank` dimension of the latent dimensions which models features interactions

`lambda_w` regularization for features interactions

`lambda_v` regularization for features

`family` one of `"binomial"`, `"gaussian"`

`intercept` logical, indicates whether or not include intecept to the model

`learning_rate_v` learning rate for features

Method `partial_fit()`**:** fits/updates model

Usage:

```
FactorizationMachine$partial_fit(x, y, weights = rep(1, length(y)), ...)
```

Arguments:

- `x` input sparse matrix. Native format is `Matrix::RsparseMatrix`. If `x` is in different format, model will try to convert it to `RsparseMatrix` with `as(x, "RsparseMatrix")`. Dimensions should be (n_samples, n_features)
- `y` vector of targets
- `weights` numeric vector of length ‘n_samples‘. Defines how to amplify SGD updates for each sample. May be useful for highly unbalanced problems.
- `...` not used at the moment

Method `fit()`: shorthand for applying ‘partial_fit‘ ‘n_iter‘ times

Usage:

```
FactorizationMachine$fit(x, y, weights = rep(1, length(y)), n_iter = 1L, ...)
```

Arguments:

- `x` input sparse matrix. Native format is `Matrix::RsparseMatrix`. If `x` is in different format, model will try to convert it to `RsparseMatrix` with `as(x, "RsparseMatrix")`. Dimensions should be (n_samples, n_features)
- `y` vector of targets
- `weights` numeric vector of length ‘n_samples‘. Defines how to amplify SGD updates for each sample. May be useful for highly unbalanced problems.
- `n_iter` number of SGD epochs
- `...` not used at the moment

Method `predict()`: makes predictions based on fitted model

Usage:

```
FactorizationMachine$predict(x, ...)
```

Arguments:

- `x` input sparse matrix of shape *(n_samples, n_featires)*
- `...` not used at the moment

Method `clone()`: The objects of this class are cloneable with this method.

Usage:

```
FactorizationMachine$clone(deep = FALSE)
```

Arguments:

- `deep` Whether to make a deep clone.

Examples

```
# Factorization Machines can fit XOR function!
x = rbind(
  c(0, 0),
  c(0, 1),
  c(1, 0),
  c(1, 1)
)
y = c(0, 1, 1, 0)

x = as(x, "RsparseMatrix")
```

```
fm = FactorizationMachine$new(learning_rate_w = 10, rank = 2, lambda_w = 0,
  lambda_v = 0, family = 'binomial', intercept = TRUE)
res = fm$fit(x, y, n_iter = 100)
preds = fm$predict(x)
all(preds[c(1, 4)] < 0.01)
all(preds[c(2, 3)] > 0.99)
```

FTRL *Logistic regression model with FTRL proximal SGD solver.*

Description

Creates 'Follow the Regularized Leader' model. Only logistic regression implemented at the moment.

Methods

Public methods:

- `FTRL$new()`
- `FTRL$partial_fit()`
- `FTRL$fit()`
- `FTRL$predict()`
- `FTRL$coef()`
- `FTRL$clone()`

Method `new()`: creates a model

Usage:

```
FTRL$new(
  learning_rate = 0.1,
  learning_rate_decay = 0.5,
  lambda = 0,
  l1_ratio = 1,
  dropout = 0,
  family = c("binomial")
)
```

Arguments:

`learning_rate` learning rate

`learning_rate_decay` learning rate which controls decay. Please refer to FTRL proximal paper for details. Usually convergense does not heavily depend on this parameter, so default value 0.5 is safe.

`lambda` regularization parameter

`l1_ratio` controls L1 vs L2 penalty mixing. 1 = Lasso regression, 0 = Ridge regression. Elastic net is in between

`dropout` dropout - percentage of random features to exclude from each sample. Acts as regularization.

`family` a description of the error distribution and link function to be used in the model. Only `binomial` (logistic regression) is implemented at the moment.

Method `partial_fit()`**:** fits model to the data

Usage:

```
FTRL$partial_fit(x, y, weights = rep(1, length(y)), ...)
```

Arguments:

`x` input sparse matrix. Native format is `Matrix::RsparseMatrix`. If `x` is in different format, model will try to convert it to `RsparseMatrix` with `as(x, "RsparseMatrix")`. Dimensions should be (n_samples, n_features)

`y` vector of targets

`weights` numeric vector of length ‘n_samples‘. Defines how to amplify SGD updates for each sample. May be useful for highly unbalanced problems.

`...` not used at the moment

Method `fit()`**:** shorthand for applying ‘partial_fit‘ ‘n_iter‘ times

Usage:

```
FTRL$fit(x, y, weights = rep(1, length(y)), n_iter = 1L, ...)
```

Arguments:

`x` input sparse matrix. Native format is `Matrix::RsparseMatrix`. If `x` is in different format, model will try to convert it to `RsparseMatrix` with `as(x, "RsparseMatrix")`. Dimensions should be (n_samples, n_features)

`y` vector of targets

`weights` numeric vector of length ‘n_samples‘. Defines how to amplify SGD updates for each sample. May be useful for highly unbalanced problems.

`n_iter` number of SGD epochs

`...` not used at the moment

Method `predict()`**:** makes predictions based on fitted model

Usage:

```
FTRL$predict(x, ...)
```

Arguments:

`x` input matrix

`...` not used at the moment

Method `coef()`**:** returns coefficients of the regression model

Usage:

```
FTRL$coef()
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
FTRL$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
library(rsparse)
library(Matrix)
i = sample(1000, 1000 * 100, TRUE)
j = sample(1000, 1000 * 100, TRUE)
y = sample(c(0, 1), 1000, TRUE)
x = sample(c(-1, 1), 1000 * 100, TRUE)
odd = seq(1, 99, 2)
x[i %in% which(y == 1) & j %in% odd] = 1
x = sparseMatrix(i = i, j = j, x = x, dims = c(1000, 1000), repr="R")

ftrl = FTRL$new(learning_rate = 0.01, learning_rate_decay = 0.1,
lambda = 10, l1_ratio = 1, dropout = 0)
ftrl$partial_fit(x, y)

w = ftrl$coef()
head(w)
sum(w != 0)
p = ftrl$predict(x)
```

`GloVe` *Global Vectors*

Description

Creates Global Vectors matrix factorization model

Public fields

`components` represents context embeddings

`bias_i` bias term i as per paper

`bias_j` bias term j as per paper

`shuffle` `logical = FALSE` by default. Whether to perform shuffling before each SGD iteration. Generally shuffling is a good practice for SGD.

Methods

Public methods:

- `GloVe$new()`
- `GloVe$fit_transform()`
- `GloVe$get_history()`
- `GloVe$clone()`

Method `new()`**:** Creates GloVe model object

Usage:

```
GloVe$new(
  rank,
  x_max,
  learning_rate = 0.15,
  alpha = 0.75,
  lambda = 0,
  shuffle = FALSE,
  init = list(w_i = NULL, b_i = NULL, w_j = NULL, b_j = NULL)
)
```

Arguments:

`rank` desired dimension for the latent vectors

`x_max` `integer` maximum number of co-occurrences to use in the weighting function

`learning_rate` `numeric` learning rate for SGD. I do not recommend that you modify this parameter, since AdaGrad will quickly adjust it to optimal

`alpha` `numeric = 0.75` the alpha in weighting function formula : $f(x) = 1 if x > x_max; else (x/x_max)^alpha$

`lambda` `numeric = 0.0` regularization parameter

`shuffle` see `shuffle` field

`init` `list(w_i = NULL, b_i = NULL, w_j = NULL, b_j = NULL)` initialization for embeddings (w_i, w_j) and biases (b_i, b_j). `w_i, w_j` - numeric matrices, should have #rows = rank, #columns = expected number of rows (w_i) / columns(w_j) in the input matrix. `b_i, b_j` = numeric vectors, should have length of #expected number of rows(b_i) / columns(b_j) in input matrix

Method `fit_transform()`**:** fits model and returns embeddings

Usage:

```
GloVe$fit_transform(
  x,
  n_iter = 10L,
  convergence_tol = -1,
  n_threads = getOption("rsparse_omp_threads", 1L),
  ...
)
```

Arguments:

`x` An input term co-occurence matrix. Preferably in `dgTMatrix` format

`n_iter` `integer` number of SGD iterations

`convergence_tol` `numeric = -1` defines early stopping strategy. Stop fitting when one of two following conditions will be satisfied: (a) passed all iterations (b) `cost_previous_iter / cost_current_iter - 1 < convergence_tol`.

`n_threads` number of threads to use

`...` not used at the moment

Method `get_history()`**:** returns value of the loss function for each epoch

Usage:

```
GloVe$get_history()
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
GloVe$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

References

http://nlp.stanford.edu/projects/glove/

Examples

```
data('movielens100k')
co_occurence = crossprod(movielens100k)
glove_model = GloVe$new(rank = 4, x_max = 10, learning_rate = .25)
embeddings = glove_model$fit_transform(co_occurence, n_iter = 2, n_threads = 1)
embeddings = embeddings + t(glove_model$components) # embeddings + context embedings
identical(dim(embeddings), c(ncol(movielens100k), 10L))
```

`LinearFlow` *Linear-FLow model for one-class collaborative filtering*

Description

Creates *Linear-FLow* model described in Practical Linear Models for Large-Scale One-Class Collaborative Filtering. The goal is to find item-item (or user-user) similarity matrix which is **low-rank and has small Frobenius norm**. Such double regularization allows to better control the generalization error of the model. Idea of the method is somewhat similar to **Sparse Linear Methods(SLIM)** but scales to large datasets much better.

Super class

`rsparse::MatrixFactorizationRecommender` -> `LinearFlow`

Public fields

`v` right singular vector of the user-item matrix. Size is `n_items * rank`. In the paper this matrix is called **v**

Methods

Public methods:

- `LinearFlow$new()`
- `LinearFlow$fit_transform()`
- `LinearFlow$transform()`
- `LinearFlow$cross_validate_lambda()`
- `LinearFlow$clone()`

Method `new()`**:** creates Linear-FLow model with `rank` latent factors.

Usage:

```
LinearFlow$new(
  rank = 8L,
  lambda = 0,
  init = NULL,
  preprocess = identity,
  solve_right_singular_vectors = c("soft_impute", "svd")
)
```

Arguments:

`rank` size of the latent dimension

`lambda` regularization parameter

`init` initialization of the orthogonal basis.

`preprocess` `identity()` by default. User spectified function which will be applied to user-item interaction matrix before running matrix factorization (also applied during inference time before making predictions). For example we may want to normalize each row of user-item matrix to have 1 norm. Or apply `log1p()` to discount large counts.

`solve_right_singular_vectors` type of the solver for initialization of the orthogonal basis. Original paper uses SVD. See paper for details.

Method `fit_transform()`**:** performs matrix factorization

Usage:

```
LinearFlow$fit_transform(x, ...)
```

Arguments:

`x` input matrix

`...` not used at the moment

Method `transform()`**:** calculates user embeddings for the new input

Usage:

```
LinearFlow$transform(x, ...)
```

Arguments:

`x` input matrix

`...` not used at the moment

Method `cross_validate_lambda()`**:** performs fast tuning of the parameter ‘lambda‘ with warm re-starts

Usage:

```
LinearFlow$cross_validate_lambda(
  x,
  x_train,
  x_test,
  lambda = "auto@10",
  metric = "map@10",
  not_recommend = x_train,
  ...
)
```

Arguments:

`x` input user-item interactions matrix. Model performs matrix facrtorization based only on this matrix

`x_train` user-item interactions matrix. Model recommends items based on this matrix. Usually should be different from ‘x‘ to avoid overfitting

`x_test` target user-item interactions. Model will evaluate predictions against this matrix, ‘x_test‘ should be treated as future interactions.

`lambda` numeric vector - sequaence of regularization parameters. Supports special value like ‘auto@10‘. This will automatically fine a sequence of lambda of length 10. This is recommended way to check for ‘lambda‘.

`metric` a metric against which model will be evaluated for top-k recommendations. Currently only `map@k` and `ndcg@k` are supported (k can be any integer)

`not_recommend` matrix same shape as ‘x_train‘. Specifies which items to not recommend for each user.

`...` not used at the moment

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
LinearFlow$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

- http://www.bkveton.com/docs/ijcai2016.pdf

Examples

```
data("movielens100k")
train = movielens100k[1:900, ]
cv = movielens100k[901:nrow(movielens100k), ]
model = LinearFlow$new(
  rank = 10, lambda = 0,
  solve_right_singular_vectors = "svd"
)
user_emb = model$fit_transform(train)
preds = model$predict(cv, k = 10)
```

`metrics` *Ranking Metrics for Top-K Items*

Description

`ap_k` calculates **Average Precision at K** (`ap@k`). Please refer to Information retrieval wikipedia article

`ndcg_k()` calculates **Normalized Discounted Cumulative Gain at K** (`ndcg@k`). Please refer to Discounted cumulative gain

Usage

```
ap_k(predictions, actual, ...)

ndcg_k(predictions, actual, ...)
```

Arguments

`predictions`	matrix of predictions. Predctions can be defined 2 ways: 1. `predictions` = `integer` matrix with item indices (correspond to column numbers in `actual`) 2. `predictions` = `character` matrix with item identifiers (characters which correspond to `colnames(actual)`) which has attribute "indices" (`integer` matrix with item indices which correspond to column numbers in `actual`).
`actual`	sparse Matrix of relevant items. Each non-zero entry considered as relevant item. Value of the each non-zero entry considered as relevance for calculation of `ndcg@k`. It should inherit from `Matrix::sparseMatrix`. Internally `Matrix::RsparseMatrix` is used.
`...`	other arguments (not used at the moment)

Examples

```
predictions = matrix(
  c(5L, 7L, 9L, 2L),
  nrow = 1
)
actual = matrix(
  c(0, 0, 0, 0, 1, 0, 1, 0, 1, 0),
  nrow = 1
)
actual = as(actual, "RsparseMatrix")
identical(rsparse::ap_k(predictions, actual), 1)
```

---

movielens100k *MovieLens 100K Dataset*

Description

This data set consists of:

1. 100,000 ratings (1-5) from 943 users on 1682 movies.
2. Each user has rated at least 20 movies.

MovieLens data sets were collected by the GroupLens Research Project at the University of Minnesota.

Usage

```
data("movielens100k")
```

Format

A sparse column-compressed matrix (`Matrix::dgCMatrix`) with 943 rows and 1682 columns.

1. rows are users
2. columns are movies
3. values are ratings

Source

https://en.wikipedia.org/wiki/MovieLens#Datasets

`PureSVD` *PureSVD recommender model decompomposition*

Description

Creates PureSVD recommender model. Solver is based on Soft-SVD which is very similar to truncated SVD but optionally adds regularization based on nuclear norm.

Super class

`rsparse::MatrixFactorizationRecommender` -> `PureSVD`

Methods

Public methods:

- `PureSVD$new()`
- `PureSVD$fit_transform()`
- `PureSVD$transform()`
- `PureSVD$clone()`

Method `new()`**:** create PureSVD model

Usage:

```
PureSVD$new(
  rank = 10L,
  lambda = 0,
  init = NULL,
  preprocess = identity,
  method = c("svd", "impute"),
  ...
)
```

Arguments:

`rank` size of the latent dimension

`lambda` regularization parameter

`init` initialization of item embeddings `preprocess` `identity()` by default. User spectified function which will be applied to user-item interaction matrix before running matrix factorization (also applied during inference time before making predictions). For example we may want to normalize each row of user-item matrix to have 1 norm. Or apply `log1p()` to discount large counts.

`method` type of the solver for initialization of the orthogonal basis. Original paper uses SVD. See paper for details.

`...` not used at the moment

Method `fit_transform()`**:** performs matrix factorization

Usage:

```
PureSVD$fit_transform(x, n_iter = 100L, convergence_tol = 0.001, ...)
```

Arguments:

`x` input sparse user-item matrix(of class `dgCMatrix`)

`n_iter` maximum number of iterations

`convergence_tol` `numeric = -Inf` defines early stopping strategy. Stops fitting when one of two following conditions will be satisfied: (a) passed all iterations (b) relative change of Frobenious norm of the two consequent solution is less then provided `convergence_tol`.

`...` not used at the moment

Method `transform()`**:** calculates user embeddings for the new input

Usage:

```
PureSVD$transform(x, ...)
```

Arguments:

`x` input matrix

`...` not used at the moment

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
PureSVD$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
data('movielens100k')
i_train = sample(nrow(movielens100k), 900)
i_test = setdiff(seq_len(nrow(movielens100k)), i_train)
train = movielens100k[i_train, ]
test = movielens100k[i_test, ]
rank = 32
lambda = 0
model = PureSVD$new(rank = rank,  lambda = lambda)
user_emb = model$fit_transform(sign(test), n_iter = 100, convergence_tol = 0.00001)
item_emb = model$components
preds = model$predict(sign(test), k = 1500, not_recommend = NULL)
mean(ap_k(preds, actual = test))
```

`ScaleNormalize` *Re-scales input matrix proportinally to item popularity*

Description

scales input user-item interaction matrix as per eq (16) from the paper. Usage of such rescaled matrix with [PureSVD] model will be equal to running PureSVD on the scaled cosine-based inter-item similarity matrix.

Public fields

`norm` which norm model should make equal to one

`scale` how to rescale norm vector

Methods

Public methods:

- `ScaleNormalize$new()`
- `ScaleNormalize$fit()`
- `ScaleNormalize$transform()`
- `ScaleNormalize$fit_transform()`
- `ScaleNormalize$clone()`

Method `new()`: creates model

Usage:

```
ScaleNormalize$new(scale = 0.5, norm = 2, target = c("rows", "columns"))
```

Arguments:

`scale` numeric, how to rescale norm vector

`norm` numeric, which norm model should make equal to one

`target` character, defines whether rows or columns should be rescaled

Method `fit()`: fits the modes

Usage:

```
ScaleNormalize$fit(x)
```

Arguments:

`x` input sparse matrix

Method `transform()`: transforms new matrix

Usage:

```
ScaleNormalize$transform(x)
```

Arguments:

`x` input sparse matrix

Method `fit_transform()`: fits the model and transforms input

Usage:

```
ScaleNormalize$fit_transform(x)
```

Arguments:

`x` input sparse matrix

Method `clone()`: The objects of this class are cloneable with this method.

Usage:

```
ScaleNormalize$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

See EigenRec: Generalizing PureSVD for Effective and Efficient Top-N Recommendations for details.

`soft_impute` *SoftImpute/SoftSVD matrix factorization*

Description

Fit SoftImpute/SoftSVD via fast alternating least squares. Based on the paper by Trevor Hastie, Rahul Mazumder, Jason D. Lee, Reza Zadeh by "Matrix Completion and Low-Rank SVD via Fast Alternating Least Squares" - http://arxiv.org/pdf/1410.2596

Usage

```
soft_impute(
  x,
  rank = 10L,
  lambda = 0,
  n_iter = 100L,
  convergence_tol = 0.001,
  init = NULL,
  final_svd = TRUE
)

soft_svd(
  x,
  rank = 10L,
  lambda = 0,
  n_iter = 100L,
  convergence_tol = 0.001,
  init = NULL,
  final_svd = TRUE
)
```

Arguments

x — sparse matrix. Both CSR `dgRMatrix` and CSC `dgCMatrix` are supported. CSR matrix is preffered because in this case algorithm will benefit from multithreaded CSR * dense matrix products (if OpenMP is supported on your platform). On many-cores machines this reduces fitting time significantly.

rank — maximum rank of the low-rank solution.

lambda — regularization parameter for the nuclear norm

n_iter — maximum number of iterations of the algorithms

convergence_tol — convergence tolerance. Internally functions keeps track of the relative change of the Frobenious norm of the two consequent iterations. If the change is less than `convergence_tol` then the process is considered as converged and function returns result.

init — svd like object with `u, v, d` components to initialize algorithm. Algorithm benefit from warm starts. `init` could be rank up `rank` of the maximum allowed rank. If `init` has rank less than max rank it will be padded automatically.

final_svd — `logical` whether need to make final preprocessing with SVD. This is not necessary but cleans up rank nicely - hithly recommnded to leave it `TRUE`.

Value

svd-like object - `list(u, v, d)`. `u, v, d` components represent left, right singular vectors and singular values.

Examples

```
set.seed(42)
data('movielens100k')
k = 10
seq_k = seq_len(k)
m = movielens100k[1:100, 1:200]
svd_ground_true = svd(m)
svd_soft_svd = soft_svd(m, rank = k, n_iter = 100, convergence_tol = 1e-6)
m_restored_svd = svd_ground_true$u[, seq_k]  %*%
   diag(x = svd_ground_true$d[seq_k]) %*%
   t(svd_ground_true$v[, seq_k])
m_restored_soft_svd = svd_soft_svd$u %*%
  diag(x = svd_soft_svd$d) %*%
  t(svd_soft_svd$v)
all.equal(m_restored_svd, m_restored_soft_svd, tolerance = 1e-1)
```

WRMF *Weighted Regularized Matrix Factorization for collaborative filtering*

Description

Creates a matrix factorization model which is solved through Alternating Least Squares (Weighted ALS for implicit feedback). For implicit feedback see "Collaborative Filtering for Implicit Feedback Datasets" (Hu, Koren, Volinsky). For explicit feedback it corresponds to the classic model for rating matrix decomposition with MSE error. These two algorithms are proven to work well in recommender systems.

Super class

`rsparse::MatrixFactorizationRecommender` -> `WRMF`

Methods

Public methods:

- `WRMF$new()`
- `WRMF$fit_transform()`
- `WRMF$transform()`
- `WRMF$clone()`

Method `new()`**:** creates WRMF model

Usage:

```
WRMF$new(
  rank = 10L,
  lambda = 0,
  dynamic_lambda = TRUE,
  init = NULL,
  preprocess = identity,
  feedback = c("implicit", "explicit"),
  solver = c("conjugate_gradient", "cholesky", "nnls"),
  with_user_item_bias = FALSE,
  with_global_bias = FALSE,
  cg_steps = 3L,
  precision = c("double", "float"),
  ...
)
```

Arguments:

`rank` size of the latent dimension

`lambda` regularization parameter

`dynamic_lambda` whether ‘lambda‘ is to be scaled according to the number

`init` initialization of item embeddings `preprocess` `identity()` by default. User spectified function which will be applied to user-item interaction matrix before running matrix factorization (also applied during inference time before making predictions). For example we may want to normalize each row of user-item matrix to have 1 norm. Or apply `log1p()` to discount large counts. This corresponds to the "confidence" function from "Collaborative Filtering for Implicit Feedback Datasets" paper. Note that it will not automatically add +1 to the weights of the positive entries.

`feedback` `character` - feedback type - one of `c("implicit", "explicit")`

`solver` `character` - solver name. One of `c("conjugate_gradient", "cholesky", "nnls")`. Usually approximate `"conjugate_gradient"` is significantly faster and solution is on par with `"cholesky"`. `"nnls"` performs non-negative matrix factorization (NNMF) - restricts user and item embeddings to be non-negative.

`with_user_item_bias` `bool` controls if model should calculate user and item biases. At the moment only implemented for `"explicit"` feedback.

`with_global_bias` `bool` controls if model should calculate global biases (mean). At the moment only implemented for `"explicit"` feedback.

`cg_steps` `integer > 0` - max number of internal steps in conjugate gradient (if "conjugate_gradient" solver used). `cg_steps = 3` by default. Controls precision of linear equation solution at the each ALS step. Usually no need to tune this parameter

`precision` one of `c("double", "float")`. Should embedding matrices be numeric or float (from `float` package). The latter is usually 2x faster and consumes less RAM. BUT `float` matrices are not "base" objects. Use carefully.

`...` not used at the moment

Method `fit_transform()`: fits the model

Usage:

```
WRMF$fit_transform(
  x,
  n_iter = 10L,
  convergence_tol = ifelse(private$feedback == "implicit", 0.005, 0.001),
  ...
)
```

Arguments:

`x` input matrix (preferably matrix in CSC format -‘CsparseMatrix‘

`n_iter` max number of ALS iterations

`convergence_tol` convergence tolerance checked between iterations

`...` not used at the moment

Method `transform()`: create user embeddings for new input

Usage:

```
WRMF$transform(x, ...)
```

Arguments:

`x` user-item iteraction matrix (preferrably as ‘dgRMatrix‘)

`...` not used at the moment

Method `clone()`: The objects of this class are cloneable with this method.

Usage:

```
WRMF$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

- Hu, Yifan, Yehuda Koren, and Chris Volinsky. "Collaborative filtering for implicit feedback datasets." 2008 Eighth IEEE International Conference on Data Mining. Ieee, 2008.
- https://math.stackexchange.com/questions/1072451/analytic-solution-for-matrix-factorization-us 1073170#1073170
- http://activisiongamescience.github.io/2016/01/11/Implicit-Recommender-Systems-Biased-Matrix-F
- https://jessesw.com/Rec-System/
- http://www.benfrederickson.com/matrix-factorization/
- http://www.benfrederickson.com/fast-implicit-matrix-factorization/
- Franc, Vojtech, Vaclav Hlavac, and Mirko Navara. "Sequential coordinate-wise algorithm for the non-negative least squares problem." International Conference on Computer Analysis of Images and Patterns. Springer, Berlin, Heidelberg, 2005.
- Zhou, Yunhong, et al. "Large-scale parallel collaborative filtering for the netflix prize." International conference on algorithmic applications in management. Springer, Berlin, Heidelberg, 2008.

Examples

```
data('movielens100k')
train = movielens100k[1:900, ]
cv = movielens100k[901:nrow(movielens100k), ]
model = WRMF$new(rank = 5,  lambda = 0, feedback = 'implicit')
user_emb = model$fit_transform(train, n_iter = 5, convergence_tol = -1)
item_emb = model$components
preds = model$predict(cv, k = 10, not_recommend = cv)
```

Index

∗ **datasets**
movielens100k, 12

ap_k (metrics), 11

detect_number_omp_threads, 2

FactorizationMachine, 3
FTRL, 5

GloVe, 7

LinearFlow, 9

metrics, 11
movielens100k, 12

ndcg_k (metrics), 11

PureSVD, 13

rsparse::MatrixFactorizationRecommender, *9*, *13*, *18*

ScaleNormalize, 15
soft_impute, 16
soft_svd (soft_impute), 16
svd, *17*

WRMF, 18