Package 'cmfrec'

May 8, 2026

Type Package

Title Collective Matrix Factorization for Recommender Systems

Version 3.5.1-3

Maintainer David Cortes <david.cortes.rivera@gmail.com>

URL https://github.com/david-cortes/cmfrec

BugReports https://github.com/david-cortes/cmfrec/issues

Description Collective matrix factorization (a.k.a. multi-view or multi-way factorization, Singh, Gordon, (2008) <doi:10.1145/1401890.1401969>) tries to approximate a (potentially very sparse
or having many missing values) matrix 'X' as the product of two low-dimensional matrices, optionally aided with secondary information matrices about rows and/or columns of 'X', which are also factorized using the same latent components.
The intended usage is for recommender systems, dimensionality reduction, and missing value imputation.
Implements extensions of the original model (Cortes, (2018) <doi:10.48550/arXiv.1809.00366>) and can produce different factorizations such as the weighted 'implicit-feedback' model (Hu, Koren, Volinsky, (2008) <doi:10.1109/ICDM.2008.22>), the 'weighted-lambda-regularization' model, (Zhou, Wilkinson, Schreiber, Pan, (2008) <doi:10.1007/978-3-540-68880-8_32>), or the enhanced model with 'implicit features' (Rendle, Zhang, Koren, (2019) <doi:10.48550/arXiv.1905.01395>), with or without side information. Can use gradient-based
procedures or alternating-least squares procedures (Koren, Bell, Volinsky, (2009) <doi:10.1109/MC.2009.263>), with either a Cholesky solver, a faster conjugate gradient solver (Takacs, Pilaszy, Tikk, (2011) <doi:10.1145/2043932.2043987>), or a non-negative coordinate descent solver (Franc, Hlavac, Navara, (2005) <doi:10.1007/11556121_50>), providing efficient methods for sparse and dense data, and mixtures thereof.
Supports L1 and L2 regularization in the main models,
offers alternative most-popular and content-based models, and implements functionality for cold-start recommendations and imputation of 2D data.

License MIT + file LICENSE

Suggests Matrix, MatrixExtra, RhpcBLASctl, recosystem (>= 0.5), recommenderlab (>= 0.2-7), MASS, knitr, rmarkdown, kableExtra

VignetteBuilder knitr

RoxygenNote 7.2.3

NeedsCompilation yes

Author David Cortes [aut, cre, cph],
Jorge Nocedal [cph] (Copyright holder of included LBFGS library),
Naoaki Okazaki [cph] (Copyright holder of included LBFGS library),
David Blackman [cph] (Copyright holder of original Xoshiro code),
Sebastiano Vigna [cph] (Copyright holder of original Xoshiro code),
NumPy Developers [cph] (Copyright holder of formatted ziggurat tables)

Repository CRAN

Date/Publication 2023-12-09 18:30:02 UTC

Contents

`CMF.from.model.matrices`

Create a CMF model object from fitted matrices

Description

Creates a ‘CMF‘ or ‘CMF_implicit‘ model object based on fitted latent factor matrices, which might have been obtained from a different software. For example, the package ‘recosystem‘ has functionality for obtaining these matrices, but not for producing recommendations or latent factors for new users, for which this function can come in handy as it will turn such model into a ‘CMF‘ model which provides all such functionality.

This is only available for models without side information, and does not support user/item mappings.

Usage

```
CMF.from.model.matrices(
  A,
  B,
  glob_mean = 0,
  implicit = FALSE,
  precompute = TRUE,
  user_bias = NULL,
  item_bias = NULL,
  lambda = 10,
  scale_lam = FALSE,
  l1_lambda = 0,
  nonneg = FALSE,
  NA_as_zero = FALSE,
  scaling_biasA = NULL,
  scaling_biasB = NULL,
  apply_log_transf = FALSE,
  alpha = 1,
  nthreads = parallel::detectCores()
)
```

Arguments

A	The obtained user factors (numeric matrix). Dimension is [k, n_users].
B	The obtained item factors (numeric matrix). Dimension is [k, n_items].
glob_mean	The obtained global mean, if the model is for explicit feedback and underwent centering. If passing zero, will assume that the values are not to be centered.
implicit	Whether this is an implicit-feedback model.
precompute	Whether to generate pre-computed matrices which can help to speed up computations on new data (see fit_models for more details).
user_bias	The obtained user biases (numeric vector). If passing 'NULL', will assume that the model did not include user biases. Dimension is [n_users].
item_bias	The obtained item biases (numeric vector). If passing 'NULL', will assume that the model did not include item biases. Dimension is [n_item].
lambda	Regularization parameter for the L2 norm of the model matrices (see fit_models for more details). Can pass different parameters for each.
scale_lam	In the explicit-feedback models, whether to scale the regularization parameter according to the number of entries. This should always be assumed 'TRUE' for models that are fit through stochastic procedures.
l1_lambda	Regularization parameter for the L1 norm of the model matrices. Same format as for 'lambda'.
nonneg	Whether the model matrices should be constrained to be non-negative.
NA_as_zero	When passing sparse matrices, whether to take missing entries as zero (counting them towards the optimization objective), or to ignore them.

`scaling_biasA` If passing it, will assume that the model uses the option 'scale_bias_const=TRUE', and will use this number as scaling for the regularization of the user biases.

`scaling_biasB` If passing it, will assume that the model uses the option 'scale_bias_const=TRUE', and will use this number as scaling for the regularization of the item biases.

`apply_log_transf` If passing 'implicit=TRUE', whether to apply a logarithm transformation on the values of 'X'.

`alpha` If passing 'implicit=TRUE', multiplier to apply to the confidence scores given by 'X'.

`nthreads` Number of parallel threads to use for further computations.

Value

A 'CMF' (if passing 'implicit=FALSE') or 'CMF_implicit' (if passing 'implicit=TRUE') model object without side information, for which the usual prediction functions such as topN and topN_new can be used as if it had been fitted through this software.

Examples

```
### Example 'adopting' a model from 'recosystem'
library(cmfrec)
library(recosystem)
library(recommenderlab)
library(MatrixExtra)

### Fitting a model with 'recosystem'
data("MovieLense")
X <- as.coo.matrix(MovieLense@data)
r <- Reco()
r$train(data_matrix(X),
        out_model = NULL,
        opts = list(dim=10, costp_l2=0.1, costq_l2=0.1,
                    verbose=FALSE, nthread=1))
matrices <- r$output(out_memory(), out_memory())
glob_mean <- as.numeric(r$model$matrices$b)

### Now converting it to CMF
model <- CMF.from.model.matrices(
    A=t(matrices$P), B=t(matrices$Q),
    glob_mean=glob_mean,
    lambda=0.1, scale_lam=TRUE,
    implicit=FALSE, nonneg=FALSE,
    nthreads=1
)

### Make predictions about new users
factors_single(model, X[10,,drop=TRUE])
topN_new(model,
         X=X[10,,drop=TRUE],
         exclude=X[10,,drop=TRUE])
```

cmfrec *cmfrec package*

Description

This is a library for approximate low-rank matrix factorizations, mainly oriented towards recommender systems but being also usable for other domains such as dimensionality reduction or imputation of missing data.

In short, the main goal behind the models in this library is to produce an approximate factorization of a matrix $\mathbf{X}$ (which might potentially be sparse or very large) as the product of two lower-dimensional matrices:
$\mathbf{X} \approx \mathbf{A}\mathbf{B}^T$
For recommender systems, it is assumed that $\mathbf{X}$ is a matrix encoding user-item interactions, with rows representing users, columns representing items, and values representing some interaction or rating (e.g. numer of times each user listened to different songs), where higher numbers mean better affinity for a given item. Under this setting, the items for which a user might have higher affinity should have larger values in the approximate factorization, and thus items with larger values for a given user can be considered as better candidates to recommend - the idea being to recommend new items to users that have not yet been seen/consumed/bought/etc.

This matrix factorization might optionally be enhanced by incorporating side information about users/items in the 'X' matrix being factorized, which is assumed to come in the form of additional matrices $\mathbf{U}$ (for users) and/or $\mathbf{I}$ (for items), which might also get factorized along the way sharing some of the same low-dimensional components.

The main function in the library is CMF, which can fit many variations of the model described above under different settings (e.g. whether entries missing in a sparse matrix are to be taken as zeros or ignored in the objective function, whether to apply centering, to including row/column intercepts, which kind of regularization to apply, etc.).

A specialized function for implicit-feedback models is also available under CMF_implicit, which provides more appropriate defaults for such data.

Nomenclature used throughout the library

The documentation and function namings use the following naming conventions:

- About data:
  - 'X' -> data about interactions between users/rows and items/columns (e.g. ratings given by users to items).
  - 'U' -> data about user/row attributes (e.g. user's age).
  - 'I' -> data about item/column attributes (e.g. a movie's genre).
- About functionalities:
  - 'warm' -> predictions based on new, unseen 'X' data, and potentially including new 'U' data along.
  - 'cold' -> predictions based on new user attributes data 'U', without 'X'.
  - 'new' -> predictions about new items based on attributes data 'I'.

- About function descriptions:
  - 'existing' -> the user/item was present in the training data to which the model was fit.
  - 'new' -> the user/items was not present in the training data that was passed to 'fit'.

Be aware that the package's functions are user-centric (e.g. it will recommend items for users, but not users for items). If predictions about new items are desired, it's recommended to use the method swap.users.and.items, as the item-based functions which are provided for convenience might run a lot slower than their user equivalents.

Implicit and explicit feedback

In recommender systems, data might come in the form of explicit user judgements about items (e.g. movie ratings) or in the form of logged user activity (e.g. number of times that a user listened to each song in a catalogue). The former is typically referred to as "explicit feedback", while the latter is referred to as "implicit feedback".

Historically, driven by the Netflix competition, formulations of this problem have geared towards predicting the rating that users would give to items under explicit-feedback datasets, determining the components in the low-rank factorization in a way that minimizes the deviation between predicted and observed numbers on the **observed** data only (i.e. predictions about items that a user did not rate do not play any role in the optimization problem for determining the low-rank factorization as they are simply ignored), but this approach has turned out to oftentimes result in very low-quality recommendations, particularly for users with few data, and is usually not suitable for implicit feedback as the data in such case does not contain any examples of dislikes and might even come in just binary (yes/no) form.

As such, research has mostly shifted towards the implicit-feedback setting, in which items that are not consumed by users do play a role in the optimization objective for determining the low-rank components - that is, the goal is more to predict which items would users have consumed than to predict the exact rating that they'd give to them - and the evaluation of recommendation quality has shifted towards looking at how items that were consumed by users would be ranked compared to unconsumed items (evaluation metrics for implicit-feedback for this library can be calculated through the package recometrics).

Other problem domains

The documentation and naming conventions in this library are all oriented towards recommender systems, with the assumption that users are rows in a matrix, items are columns, and values denote interactions between them, with the idea that values under different columns are comparable (e.g. the rating scale is the same for all items).

The concept of approximate low-rank matrix factorizations is however still useful for other problem domains, such as general dimensionality reduction for large sparse data (e.g. TF-IDF matrices) or imputation of high-dimensional tabular data, in which assumptions like values being comparable between different columns would not hold.

Be aware that functions like CMF come with some defaults that might not be reasonable in other applications, but which can be changed by passing non-default arguments to functions - for example:

- Global centering - the "explicit-feedback" models here will by default calculate a global mean for all entries in 'X' and center the matrix by substracting this value from all entries. This is a reasonable thing to do when dealing with movie ratings as all ratings follow the same scale,

but if columns of the 'X' matrix represent different things that might have different ranges or different distributions, global mean centering is probably not going to be desirable or useful.

- User/row biases: models might also have one bias/intercept parameter per row, which in the approximation, would get added to every column for that user/row. This is again a reasonable thing to do for movie ratings, but if the columns of 'X' contain different types of information, it might not be a sensible thing to add.
- Regularization for item/column biases: since the models perform global mean centering beforehand, the item/column-specific bias/intercept parameters will get a regularization penalty ("shrinkage") applied to them, which might not be desirable if global mean centering is removed.

Improving performance

This library will run faster when compiled from source with non-default compiler arguments, particularly '-march=native' (replace with '-mcpu=native' for ARM/PPC); and when using an optimized BLAS library for R. See this guide for details: installing optimized libraries.

Author(s)

Maintainer: David Cortes <david.cortes.rivera@gmail.com> [copyright holder]

Other contributors:

- Jorge Nocedal (Copyright holder of included LBFGS library) [copyright holder]
- Naoaki Okazaki (Copyright holder of included LBFGS library) [copyright holder]
- David Blackman (Copyright holder of original Xoshiro code) [copyright holder]
- Sebastiano Vigna (Copyright holder of original Xoshiro code) [copyright holder]
- NumPy Developers (Copyright holder of formatted ziggurat tables) [copyright holder]

See Also

CMF CMF_implicit

---

drop.nonessential.matrices

Drop matrices that are not used for prediction

Description

Drops all the matrices in the model object which are not used for calculating new user factors (either warm or cold), such as the user biases or the item factors.

This is intended at decreasing memory usage in production systems which use this software for calculation of user factors or top-N recommendations.

Can additionally drop some of the precomputed matrices which are only taken in special circumstances such as when passing dense data with no missing values - however, predictions that would have otherwise used these matrices will become slower afterwards.

After dropping these non-essential matrices, it will not be possible anymore to call certain methods such as ‘predict‘ or ‘swap.users.and.items‘. The methods which are intended to continue working afterwards are:

- factors_single
- factors
- topN_new

This method is only available for ‘CMF‘ and ‘CMF_implicit‘ model objects.

Usage

```
drop.nonessential.matrices(model, drop_precomputed = TRUE)
```

Arguments

`model`	A model object as returned by CMF or CMF_implicit.
`drop_precomputed`	Whether to drop the less commonly used prediction matrices (see documentation above for more details).

Details

After calling this function and reassigning the output to the original model object, one might need to call the garbage collector (by running ‘gc()‘) before any of the freed memory is shown as available.

Value

The model object with the non-essential matrices dropped.

`factors` *Calculate latent factors on new data*

Description

Determine latent factors for new user(s)/row(s), given either ‘X‘ data (a.k.a. "warm-start"), or ‘U‘ data (a.k.a. "cold-start"), or both.

If passing both types of data (‘X‘ and ‘U‘), and the number of rows in them differs, will be assumed that the shorter matrix has only missing values for the unmatched entries in the other matrix.

Note: this function will not perform any internal re-indexing for the data. If the ‘X‘ to which the data was fit was a ‘data.frame‘, the numeration of the items will be under ‘model$info$item_mapping‘. There is also a function factors_single which will let the model do the appropriate reindexing.

For example usage, see the main section fit_models.

Usage

```
factors(model, ...)

## S3 method for class 'CMF'
factors(
  model,
  X = NULL,
  U = NULL,
  U_bin = NULL,
  weight = NULL,
  output_bias = FALSE,
  nthreads = model$info$nthreads,
  ...
)

## S3 method for class 'CMF_implicit'
factors(model, X = NULL, U = NULL, nthreads = model$info$nthreads, ...)

## S3 method for class 'ContentBased'
factors(model, U, nthreads = model$info$nthreads, ...)

## S3 method for class 'OMF_explicit'
factors(
  model,
  X = NULL,
  U = NULL,
  weight = NULL,
  output_bias = FALSE,
  output_A = FALSE,
  exact = FALSE,
  nthreads = model$info$nthreads,
  ...
)

## S3 method for class 'OMF_implicit'
factors(
  model,
  X = NULL,
  U = NULL,
  output_A = FALSE,
  nthreads = model$info$nthreads,
  ...
)
```

Arguments

model	A collective matrix factorization model from this package - see fit_models for details.

`...` Not used.

`X` New ‘X‘ data, with rows denoting new users. Can be passed in the following formats:

- A sparse COO/triplets matrix, either from package ‘Matrix‘ (class ‘dgTMatrix‘), or from package ‘SparseM‘ (class ‘matrix.coo‘).
- A sparse matrix in CSR format, either from package ‘Matrix‘ (class ‘dgRMatrix‘), or from package ‘SparseM‘ (class ‘matrix.csr‘). Passing the input as CSR is faster than COO as it will be converted internally.
- A sparse row vector from package ‘Matrix‘ (class ‘dsparseVector‘).
- A dense matrix from base R (class ‘matrix‘), with missing entries set as ‘NA‘/‘NaN‘.
- A dense row vector from base R (class ‘numeric‘), with missing entries set as ‘NA‘/‘NaN‘.

Dense ‘X‘ data is not supported for ‘CMF_implicit‘ or ‘OMF_implicit‘.

`U` New ‘U‘ data, with rows denoting new users. Can be passed in the same formats as ‘X‘, or additionally as a ‘data.frame‘, which will be internally converted to a matrix.

`U_bin` New binary columns of ‘U‘. Must be passed as a dense matrix from base R or as a ‘data.frame‘.

`weight` Associated observation weights for entries in ‘X‘. If passed, must have the same shape as ‘X‘ - that is, if ‘X‘ is a sparse matrix, should be a numeric vector with length equal to the non-missing elements (or a sparse matrix in the same format, but will not make any checks on the indices), if ‘X‘ is a dense matrix, should also be a dense matrix with the same number of rows and columns.

`output_bias` Whether to also return the user bias determined by the model given the data in ‘X‘.

`nthreads` Number of parallel threads to use.

`output_A` Whether to return the raw ‘A‘ factors (the free offset).

`exact` (In the ‘OMF_explicit‘ model) Whether to calculate ‘A‘ and ‘Am‘ with the regularization applied to ‘A‘ instead of to ‘Am‘ (if using the L-BFGS method, this is how the model was fit). This is usually a slower procedure. Only relevant when passing ‘X‘ data.

Details

Note that, regardless of whether the model was fit with the L-BFGS or ALS method with CG or Cholesky solver, the new factors will be determined through the Cholesky method or through the precomputed matrices (e.g. a simple matrix-matrix multiply for the ‘ContentBased‘ model), unless passing ‘U_bin‘ in which case they will be determined through the same L-BFGS method with which the model was fit.

Value

If passing ‘output_bias=FALSE‘, ‘output_A=FALSE‘, and for the implicit-feedback models, will return a matrix with the obtained latent factors for each row/user given the ‘X‘ and/or ‘U‘ data (number of rows is ‘max(nrow(X), nrow(U), nrow(U_bin))‘). If passing any of the above options, will return a list with the following elements:

- ‘factors‘: The obtained latent factors (a matrix).
- ‘bias‘: (If passing ‘output_bias=TRUE‘) A vector with the obtained biases for each row/user.
- ‘A‘: (If passing ‘output_A=TRUE‘) The raw ‘A‘ factors matrix (which is added to the factors determined from user attributes in order to obtain the factorization parameters).

See Also

factors_single

`factors_single` *Calculate latent factors for a new user*

Description

Determine latent factors for a new user, given either ‘X‘ data (a.k.a. "warm-start"), or ‘U‘ data (a.k.a. "cold-start"), or both.

For example usage, see the main section fit_models.

Usage

```
factors_single(model, ...)

## S3 method for class 'CMF'
factors_single(
  model,
  X = NULL,
  X_col = NULL,
  X_val = NULL,
  U = NULL,
  U_col = NULL,
  U_val = NULL,
  U_bin = NULL,
  weight = NULL,
  output_bias = FALSE,
  ...
)

## S3 method for class 'CMF_implicit'
factors_single(
  model,
  X = NULL,
  X_col = NULL,
  X_val = NULL,
  U = NULL,
  U_col = NULL,
  U_val = NULL,
```

```
  ...
)

## S3 method for class 'ContentBased'
factors_single(model, U = NULL, U_col = NULL, U_val = NULL, ...)

## S3 method for class 'OMF_explicit'
factors_single(
  model,
  X = NULL,
  X_col = NULL,
  X_val = NULL,
  U = NULL,
  U_col = NULL,
  U_val = NULL,
  weight = NULL,
  output_bias = FALSE,
  output_A = FALSE,
  exact = FALSE,
  ...
)

## S3 method for class 'OMF_implicit'
factors_single(
  model,
  X = NULL,
  X_col = NULL,
  X_val = NULL,
  U = NULL,
  U_col = NULL,
  U_val = NULL,
  output_A = FALSE,
  ...
)
```

Arguments

model	A collective matrix factorization model from this package - see fit_models for details.
...	Not used.
X	New ‘X‘ data, either as a numeric vector (class ‘numeric‘), or as a sparse vector from package ‘Matrix‘ (class ‘dsparseVector‘). If the ‘X‘ to which the model was fit was a ‘data.frame‘, the column/item indices will have been reindexed internally, and the numeration can be found under ‘model$info$item_mapping‘. Alternatively, can instead pass the column indices and values and let the model reindex them (see ‘X_col‘ and ‘X_val‘). Should pass at most one of ‘X‘ or ‘X_col‘+‘X_val‘. Dense ‘X‘ data is not supported for ‘CMF_implicit‘ or ‘OMF_implicit‘.
X_col	New ‘X‘ data in sparse vector format, with ‘X_col‘ denoting the items/columns

which are not missing. If the ‘X‘ to which the model was fit was a ‘data.frame‘, here should pass IDs matching to the second column of that ‘X‘, which will be reindexed internally. Otherwise, should have column indices with numeration starting at 1 (passed as an integer vector). Should pass at most one of ‘X‘ or ‘X_col‘+‘X_val‘.

`X_val` New ‘X‘ data in sparse vector format, with ‘X_val‘ denoting the associated values to each entry in ‘X_col‘ (should be a numeric vector of the same length as ‘X_col‘). Should pass at most one of ‘X‘ or ‘X_col‘+‘X_val‘.

`U` New ‘U‘ data, either as a numeric vector (class ‘numeric‘), or as a sparse vector from package ‘Matrix‘ (class ‘dsparseVector‘). Alternatively, if ‘U‘ is sparse, can instead pass the indices of the non-missing columns and their values separately (see ‘U_col‘). Should pass at most one of ‘U‘ or ‘U_col‘+‘U_val‘.

`U_col` New ‘U‘ data in sparse vector format, with ‘U_col‘ denoting the attributes/columns which are not missing. Should have numeration starting at 1 (should be an integer vector). Should pass at most one of ‘U‘ or ‘U_col‘+‘U_val‘.

`U_val` New ‘U‘ data in sparse vector format, with ‘U_val‘ denoting the associated values to each entry in ‘U_col‘ (should be a numeric vector of the same length as ‘U_col‘). Should pass at most one of ‘U‘ or ‘U_col‘+‘U_val‘.

`U_bin` Binary columns of ‘U‘ on which a sigmoid transformation will be applied. Should be passed as a numeric vector. Note that ‘U‘ and ‘U_bin‘ are not mutually exclusive.

`weight` (Only for the explicit-feedback models) Associated weight to each non-missing observation in ‘X‘. Must have the same number of entries as ‘X‘ - that is, if passing a dense vector of length ‘n‘, ‘weight‘ should be a numeric vector of length ‘n‘ too, if passing a sparse vector, should have a length corresponding to the number of non-missing elements. or alternatively, may be a sparse matrix/vector with the same non-missing indices as ‘X‘ (but this will not be checked).

`output_bias` Whether to also return the user bias determined by the model given the data in ‘X‘.

`output_A` Whether to return the raw ‘A‘ factors (the free offset).

`exact` (In the ‘OMF_explicit‘ model) Whether to calculate ‘A‘ and ‘Am‘ with the regularization applied to ‘A‘ instead of to ‘Am‘ (if using the L-BFGS method, this is how the model was fit). This is usually a slower procedure. Only relevant when passing ‘X‘ data.

Details

Note that, regardless of whether the model was fit with the L-BFGS or ALS method with CG or Cholesky solver, the new factors will be determined through the Cholesky method or through the precomputed matrices (e.g. a simple matrix-vector multiply for the ‘ContentBased‘ model), unless passing ‘U_bin‘ in which case they will be determined through the same L-BFGS method with which the model was fit.

Value

If passing ‘output_bias=FALSE‘, ‘output_A=FALSE‘, and in the implicit-feedback models, will return a vector with the obtained latent factors. If passing any of the earlier options, will return a

list with the following entries:

- ‘factors‘, which will contain the obtained factors for this new user.
- ‘bias‘, which will contain the obtained bias for this new user (if passing ‘output_bias=TRUE‘) (this will be a single number).
- ‘A‘ (if passing ‘output_A=TRUE‘), which will contain the raw ‘A‘ vector (which is added to the factors determined from user attributes in order to obtain the factorization parameters).

See Also

factors topN_new

`fit_models` *Matrix Factorization Models*

Description

Models for collective matrix factorization (also known as multi-view or multi-way). These models try to approximate a matrix ‘X‘ as the product of two lower-rank matrices ‘A‘ and ‘B‘ (that is: $\mathbf{X} \approx \mathbf{A}\mathbf{B}^T$) by finding the values of ‘A‘ and ‘B‘ that minimize the squared error w.r.t. ‘X‘, optionally aided with side information matrices ‘U‘ and ‘I‘ about rows and columns of ‘X‘.

The package documentation is built with recommendation systems in mind, for which it assumes that ‘X‘ is a sparse matrix in which users represent rows, items represent columns, and the non-missing values denote interactions such as movie ratings from users to items. The idea behind it is to recommend the missing entries in ‘X‘ that have the highest predicted value according to the approximation. For other domains, take any mention of users as rows and any mention of items as columns (e.g. when used for topic modeling, the "users" are documents and the "items" are word occurrences).

In the ‘CMF‘ model (main functionality of the package and most flexible model type), the ‘A‘ and ‘B‘ matrices are also used to jointly factorize the side information matrices - that is: $\mathbf{U} \approx \mathbf{A}\mathbf{C}^T$, $\mathbf{I} \approx \mathbf{B}\mathbf{D}^T$, sharing the same components or latent factors for two factorizations. Informally, this means that the obtained factors now need to explain both the interactions data and the attributes data, making them generalize better to the non-present entries of ‘X‘ and to new data.

In ‘CMF‘ and the other non-implicit models, the ‘X‘ data is always centered beforehand by subtracting its mean, and might optionally add user and item biases (which are model parameters, not pre-estimated).

The model might optionally generate so-called implicit features from the same ‘X‘ data, by factorizing binary matrices which tell which entries in ‘X‘ are present, i.e.: $\mathbf{I}_x \approx \mathbf{A}\mathbf{B}_i^T$, $\mathbf{I}_x^T \approx \mathbf{B}\mathbf{A}_i^T$, where $\mathbf{I}_x$ is an indicator matrix which is treated as full (no unknown values).

The ‘CMF_implicit‘ model extends the collective factorization idea to the implicit-feedback case, based on reference [3]. While in ‘CMF‘ the values of ‘X‘ are taken at face value and the objective is to minimize squared error over the non-missing entries, in the implicit-feedback variants the matrix ‘X‘ is assumed to be binary (all entries are zero or one, with no unknown values), with the positive entries (those which are not missing in the data) having a weight determined by ‘X‘, and without including any user/item biases or centering for the ’X’ matrix.

'CMF' is intended for explicit feedback data (e.g. movie ratings, which contain both likes and dislikes), whereas 'CMF_implicit' is intended for implicit feedback data (e.g. number of times each user watched each movie/series, which do not contain dislikes and the values are treated as confidence scores).

The 'MostPopular' model is a simpler heuristic implemented for comparison purposes which is equivalent to either 'CMF' or 'CMF_implicit' with 'k=0' plus user/item biases. If a personalized model is not able to beat this heuristic under the evaluation metrics of interest, chances are that such personalized model needs better tuning.

The 'ContentBased' model offers a different alternative in which the latent factors are determined directly from the user/item attributes (which are no longer optional) - that is: $\mathbf{A} = \mathbf{UC}$, $\mathbf{B} = \mathbf{ID}$, optionally adding per-column intercepts, and is aimed at cold-start predictions (such a model is extremely unlikely to perform better for new users in the presence of interactions data). For this model, the package provides functionality for making predictions about potential new entries in 'X' which involve both new rows and new columns at the same time. Unlike the others, it does not offer an implicit-feedback variant.

The 'OMF_explicit' model extends the 'ContentBased' by adding a free offset determined for each user and item according to 'X' data alone - that is: $\mathbf{A}_m = \mathbf{A} + \mathbf{UC}$, $\mathbf{B}_m = \mathbf{B} + \mathbf{ID}$, $\mathbf{X} \approx \mathbf{A}_m \mathbf{B}_m^T$, and 'OMF_implicit' extends the idea to the implicit-feedback case.

Note that 'ContentBased' is equivalent to 'OMF_explicit' with 'k=0', 'k_main=0' and 'k_sec>0' (see documentation for details about these parameters). For a different formulation in which user factors are determined directly for item attributes (and same for items with user attributes), it's also possible to use 'OMF_explicit' with 'k=0' while passing 'k_sec' and 'k_main'.

('OMF_explicit' and 'OMF_implicit' were only implemented for research purposes for cold-start recommendations in cases in which there is side info about users but not about items or vice-versa - it is not recommended to rely on them.)

Some extra considerations about the parameters here:

- By default, the terms in the optimization objective are not scaled by the number of entries (see parameter 'scale_lam'), thus hyperparameters such as 'lambda' will require more tuning than in other software and will require trying a wider range of values.
- The regularization applied to the matrices is the same for all users and for all items.
- The default hyperparameters are not geared towards speed - for faster fitting times, use 'method='als'', 'use_cg=TRUE', 'finalize_chol=FALSE', 'precompute_for_predictions=FALSE', 'verbose=FALSE', and pass 'X' as a matrix (either sparse or dense).
- The default hyperparameters are also very different than in other software - for example, for 'CMF_implicit', in order to match the Python package's 'implicit' hyperparameters, one would have to use 'k=100', 'lambda=0.01', 'niter=15', 'use_cg=TRUE', 'finalize_chol=FALSE', and use single-precision floating point numbers (not supported in the R version of this package).

Usage

```
CMF(
  X,
  U = NULL,
  I = NULL,
```

```
  U_bin = NULL,
  I_bin = NULL,
  weight = NULL,
  k = 40L,
  lambda = 10,
  method = "als",
  use_cg = TRUE,
  user_bias = TRUE,
  item_bias = TRUE,
  center = TRUE,
  add_implicit_features = FALSE,
  scale_lam = FALSE,
  scale_lam_sideinfo = FALSE,
  scale_bias_const = FALSE,
  k_user = 0L,
  k_item = 0L,
  k_main = 0L,
  w_main = 1,
  w_user = 1,
  w_item = 1,
  w_implicit = 0.5,
  l1_lambda = 0,
  center_U = TRUE,
  center_I = TRUE,
  maxiter = 800L,
  niter = 10L,
  parallelize = "separate",
  corr_pairs = 4L,
  max_cg_steps = 3L,
  precondition_cg = FALSE,
  finalize_chol = TRUE,
  NA_as_zero = FALSE,
  NA_as_zero_user = FALSE,
  NA_as_zero_item = FALSE,
  nonneg = FALSE,
  nonneg_C = FALSE,
  nonneg_D = FALSE,
  max_cd_steps = 100L,
  precompute_for_predictions = TRUE,
  include_all_X = TRUE,
  verbose = TRUE,
  print_every = 10L,
  handle_interrupt = TRUE,
  seed = 1L,
  nthreads = parallel::detectCores()
)

CMF_implicit(
```

```
  X,
  U = NULL,
  I = NULL,
  k = 40L,
  lambda = 1,
  alpha = 1,
  use_cg = TRUE,
  k_user = 0L,
  k_item = 0L,
  k_main = 0L,
  w_main = 1,
  w_user = 1,
  w_item = 1,
  l1_lambda = 0,
  center_U = TRUE,
  center_I = TRUE,
  niter = 10L,
  max_cg_steps = 3L,
  precondition_cg = FALSE,
  finalize_chol = FALSE,
  NA_as_zero_user = FALSE,
  NA_as_zero_item = FALSE,
  nonneg = FALSE,
  nonneg_C = FALSE,
  nonneg_D = FALSE,
  max_cd_steps = 100L,
  apply_log_transf = FALSE,
  precompute_for_predictions = TRUE,
  verbose = TRUE,
  handle_interrupt = TRUE,
  seed = 1L,
  nthreads = parallel::detectCores()
)

MostPopular(
  X,
  weight = NULL,
  implicit = FALSE,
  center = TRUE,
  user_bias = ifelse(implicit, FALSE, TRUE),
  lambda = 10,
  alpha = 1,
  NA_as_zero = FALSE,
  apply_log_transf = FALSE,
  nonneg = FALSE,
  scale_lam = FALSE,
  scale_bias_const = FALSE
)
```

```
ContentBased(
  X,
  U,
  I,
  weight = NULL,
  k = 20L,
  lambda = 100,
  user_bias = FALSE,
  item_bias = FALSE,
  add_intercepts = TRUE,
  maxiter = 3000L,
  corr_pairs = 3L,
  parallelize = "separate",
  verbose = TRUE,
  print_every = 100L,
  handle_interrupt = TRUE,
  start_with_ALS = TRUE,
  seed = 1L,
  nthreads = parallel::detectCores()
)

OMF_explicit(
  X,
  U = NULL,
  I = NULL,
  weight = NULL,
  k = 50L,
  lambda = 10,
  method = "lbfgs",
  use_cg = TRUE,
  precondition_cg = FALSE,
  user_bias = TRUE,
  item_bias = TRUE,
  center = TRUE,
  k_sec = 0L,
  k_main = 0L,
  add_intercepts = TRUE,
  w_user = 1,
  w_item = 1,
  maxiter = 10000L,
  niter = 10L,
  parallelize = "separate",
  corr_pairs = 7L,
  max_cg_steps = 3L,
  finalize_chol = TRUE,
  NA_as_zero = FALSE,
  verbose = TRUE,
```

```
  print_every = 100L,
  handle_interrupt = TRUE,
  seed = 1L,
  nthreads = parallel::detectCores()
)

OMF_implicit(
  X,
  U = NULL,
  I = NULL,
  k = 50L,
  lambda = 1,
  alpha = 1,
  use_cg = TRUE,
  precondition_cg = FALSE,
  add_intercepts = TRUE,
  niter = 10L,
  apply_log_transf = FALSE,
  max_cg_steps = 3L,
  finalize_chol = FALSE,
  verbose = FALSE,
  handle_interrupt = TRUE,
  seed = 1L,
  nthreads = parallel::detectCores()
)
```

Arguments

X

The main matrix with interactions data to factorize (e.g. movie ratings by users, bag-of-words representations of texts, etc.). The package is built with recommender systems in mind, and will assume that 'X' is a matrix in which users are rows, items are columns, and values denote interactions between a given user and item. Can be passed in the following formats:

- A 'data.frame' representing triplets, in which there should be one row for each present or non-missing interaction, with the first column denoting the user/row ID, the second column the item/column ID, and the third column the value (e.g. movie rating). If passed in this format, the user and item IDs will be reindexed internally, and the side information matrices should have row names matching to those IDs. If there are observation weights, these should be the fourth column.
- A sparse matrix in COO/triplets format, either from package 'Matrix' (class 'dgTMatrix') or from package 'SparseM' (class 'matrix.coo').
- A dense matrix from base R (class 'matrix'), with missing values set as 'NA'/'NaN'.

If using the package 'softImpute', objects of type 'incomplete' from that package can be converted to 'Matrix' objects through e.g. 'as(X, "TsparseMatrix")'. Sparse matrices can be created through e.g. 'Matrix::sparseMatrix(..., repr="T")'.

It is recommended for faster fitting times to pass the ‘X‘ data as a matrix (either sparse or dense) as then it will avoid internal reindexes.

Note that, generally, it’s possible to pass partially disjoints sets of users/items between the different matrices (e.g. it’s possible for both the ‘X‘ and ‘U‘ matrices to have rows that the other doesn’t have). If any of the inputs has less rows/columns than the other(s) (e.g. ‘U‘ has more rows than ‘X‘, or ‘I‘ has more rows than there are columns in ‘X‘), will assume that the rest of the rows/columns have only missing values. However, when having partially disjoint inputs, the order of the rows/columns matters for speed for the ‘CMF‘ and ‘CMF_implicit‘ models under the ALS method, as it might run faster when the ‘U‘/‘I‘ inputs that do not have matching rows/columns in ‘X‘ have those unmatched rows/columns at the end (last rows/columns) and the ‘X‘ input is shorter. See also the parameter ‘include_all_X‘ for info about predicting with mismatched ‘X‘.

If passed as sparse/triplets, the non-missing values should not contain any ‘NA‘/‘NaN‘s.

`U` User attributes information. Can be passed in the following formats:

- A ‘matrix‘, with rows corresponding to rows of ‘X‘ and columns to user attributes. For the ‘CMF‘ and ‘CMF_implicit‘ models, missing values are supported and should be set to ‘NA‘/‘NaN‘.
- A ‘data.frame‘ with the same format as above.
- A sparse matrix in COO/triplets format, either from package ‘Matrix‘ (class ‘dgTMatrix‘) or from package ‘SparseM‘ (class ‘matrix.coo‘). Same as above, rows correspond to rows of ‘X‘ and columns to user attributes. If passed as sparse, the non-missing values cannot contain ‘NA‘/‘NaN‘ - see parameter ‘NA_as_zero_user‘ for how to interpret non-missing values. Sparse side info is not supported for ‘OMF_implicit‘, nor for ‘OMF_explicit‘ with ‘method=als‘.

If ‘X‘ is a ‘data.frame‘, should be either a ‘data.frame‘ or ‘matrix‘, containing row names matching to the first column of ‘X‘ (which denotes the user/row IDs of the non-zero entries). If ‘U‘ is sparse, ‘X‘ should be passed as sparse or dense matrix (not a ‘data.frame‘).

Note that, if ‘U‘ is a ‘matrix‘ or ‘data.frame‘, it should have the same number of rows as ‘X‘ in the ‘ContentBased‘, ‘OMF_explicit‘, and ‘OMF_implicit‘ models.

Be aware that ‘CMF‘ and ‘CMF_implicit‘ tend to perform better with dense and not-too-wide user/item attributes.

`I` Item attributes information. Can be passed in the following formats:

- A ‘matrix‘, with rows corresponding to columns of ‘X‘ and columns to item attributes. For the ‘CMF‘ and ‘CMF_implicit‘ models, missing values are supported and should be set to ‘NA‘/‘NaN‘.
- A ‘data.frame‘ with the same format as above.
- A sparse matrix in COO/triplets format, either from package ‘Matrix‘ (class ‘dgTMatrix‘) or from package ‘SparseM‘ (class ‘matrix.coo‘). Same as above, rows correspond to columns of ‘X‘ and columns to item attributes. If passed as sparse, the non-missing values cannot contain ‘NA‘/‘NaN‘ - see parameter ‘NA_as_zero_item‘ for how to interpret non-missing values.

Sparse side info is not supported for 'OMF_implicit', nor for 'OMF_explicit' with 'method=als'.

If 'X' is a 'data.frame', should be either a 'data.frame' or 'matrix', containing row names matching to the second column of 'X' (which denotes the item/column IDs of the non-zero entries). If 'I' is sparse, 'X' should be passed as sparse or dense matrix (not a 'data.frame').

Note that, if 'I' is a 'matrix' or 'data.frame', it should have the same number of rows as there are columns in 'X' in the 'ContentBased', 'OMF_explicit', and 'OMF_implicit' models.

Be aware that 'CMF' and 'CMF_implicit' tend to perform better with dense and not-too-wide user/item attributes.

`U_bin` User binary columns/attributes (all values should be zero, one, or missing), for which a sigmoid transformation will be applied on the predicted values. If 'X' is a 'data.frame', should also be a 'data.frame', with row names matching to the first column of 'X' (which denotes the user/row IDs of the non-zero entries). Cannot be passed as a sparse matrix. Note that 'U' and 'U_bin' are not mutually exclusive.

Only supported with "method='lbfgs'".

`I_bin` Item binary columns/attributes (all values should be zero, one, or missing), for which a sigmoid transformation will be applied on the predicted values. If 'X' is a 'data.frame', should also be a 'data.frame', with row names matching to the second column of 'X' (which denotes the item/column IDs of the non-zero entries). Cannot be passed as a sparse matrix. Note that 'I' and 'I_bin' are not mutually exclusive.

Only supported with "method='lbfgs'".

`weight` (Optional and not recommended) Observation weights for entries in 'X'. Must have the same shape as 'X' - that is, if 'X' is a sparse matrix, must be a vector with the same number of non-zero entries as 'X', if 'X' is a dense matrix, 'weight' must also be a dense matrix. Alternatively, if 'X' is a sparse COO matrix, 'weight' may also be passed as a sparse COO matrix in the same format, but it will not check whether the indices match between the two. If 'X' is a 'data.frame', should be passed instead as its fourth column.

Cannot have missing values.

This is only supported for the explicit-feedback models, as the implicit-feedback ones determine the weights through 'X'.

`k` Number of latent factors to use (dimensionality of the low-rank factorization) - these will be shared between the factorization of the 'X' matrix and the side info matrices in the 'CMF' and 'CMF_implicit' models, and will be determined jointly by interactions and side info in the 'OMF_explicit' and 'OMF_implicit' models. Additional non-shared components can also be specified through 'k_user', 'k_item', and 'k_main' (also 'k_sec' for 'OMF_explicit').

Typical values are 30 to 100.

`lambda` Regularization parameter to apply on the squared L2 norms of the matrices. Some models ('CMF', 'CMF_implicit', 'ContentBased', and 'OMF_explicit' with the L-BFGS method) can use different regularization for each matrix, in

which case it should be an array with 6 entries (regardless of the model), corresponding, in this order, to: ‘user_bias‘, ‘item_bias‘, ‘A‘, ‘B‘, ‘C‘, ‘D‘. Note that the default value for ‘lambda‘ here is much higher than in other software, and that the loss/objective function is not divided by the number of entries anywhere, so this parameter needs good tuning. For example, a good value for the MovieLens10M would be ‘lambda=35‘ (or ‘lambda=0.05‘ with ‘scale_lam=TRUE‘), whereas for the LastFM-360K, a good value would be ‘lambda=5‘.

Typical values are 10^{-2} to 10^{2}, with the implicit-feedback models requiring less regularization.

`method` Optimization method used to fit the model. If passing ‘lbfgs‘, will fit it through a gradient-based approach using an L-BFGS optimizer, and if passing ‘als‘, will fit it through the ALS (alternating least-squares) method. L-BFGS is typically a much slower and a much less memory efficient method compared to ‘als‘, but tends to reach better local optima and allows some variations of the problem which ALS doesn’t, such as applying sigmoid transformations for binary side information.

Note that not all models allow choosing the optimizer:

- ‘CMF_implicit‘ and ‘OMF_implicit‘ can only be fitted through the ALS method.
- ‘ContentBased‘ can only be fitted through the L-BFGS method.
- ‘MostPopular‘ can only use an ALS-like procedure, but which will ignore parameters such as ‘niter‘.
- Models with non-negativity constraints can only be fitted through the ALS method, and the matrices to which the constraints apply can only be determined through a coordinate descent procedure (which will ignore what is passed to ‘use_cg‘ and ‘finalize_chol‘).
- Models with L1 regularization can only be fitted through the ALS method, and the sub-problems are solved through a coordinate-descent procedure.

`use_cg` In the ALS method, whether to use a conjugate gradient method to solve the closed-form least squares problems. This is a faster and more memory-efficient alternative than the default Cholesky solver, but less exact, less numerically stable, and will require slightly more ALS iterations (‘niter‘) to reach a good optimum. In general, better results are achieved with ‘use_cg=FALSE‘ for the explicit-feedback models. Note that, if using this method, calculations after fitting which involve new data such as factors, might produce slightly different results from the factors obtained inside the fitted model with the same data, due to differences in numerical precision. A workaround for this issue (factors on new data that might differ slightly) is to use ‘finalize_chol=TRUE‘. Even if passing ‘TRUE‘ here, will use the Cholesky method in cases in which it is faster (e.g. dense matrices with no missing values), and will not use the conjugate gradient method on new data. This option is not available when using L1 regularization and/or non-negativity constraints. Ignored when using the L-BFGS method.

`user_bias` Whether to add user/row biases (intercepts) to the model. If using it for purposes other than recommender systems, this is is usually **not** suggested to include.

`item_bias` Whether to add item/column biases (intercepts) to the model. Be aware that using item biases with low regularization for them will tend to favor items with high average ratings regardless of the number of ratings the item has received.

`center` Whether to center the "X" data by subtracting the mean value. For recommender systems, it's highly recommended to pass 'TRUE' here, the more so if the model has user and/or item biases.

For 'MostPopular', if passing 'implicit=TRUE', this option will be ignored (assumed 'FALSE').

`add_implicit_features` Whether to automatically add so-called implicit features from the data, as in reference [5] and similar. If using this for recommender systems with small amounts of data, it's recommended to pass 'TRUE' here.

`scale_lam` Whether to scale (increase) the regularization parameter for each row of the model matrices (A, B, C, D) according to the number of non-missing entries in the data for that particular row, as proposed in reference [7]. For the A and B matrices, the regularization will only be scaled according to the number of non-missing entries in 'X' (see also the 'scale_lam_sideinfo' parameter). Note that, when using the options 'NA_as_zero_*', all entries are considered to be non-missing. If passing 'TRUE' here, the optimal value for 'lambda' will be much smaller (and likely below 0.1). This option tends to give better results, but requires more hyperparameter tuning. Only supported for the ALS method.

For the 'MostPopular' model, this is not supported when passing 'implicit=TRUE', and it is not recommended to use for it, as it will tend to recommend items which have a single user interaction with the maximum possible value (e.g. 5-star movies from only 1 user).

When generating factors based on side information alone, if passing 'scale_lam_sideinfo', will regularize assuming there was one observation present. Be aware that using this option **without** 'scale_lam_sideinfo=TRUE' can lead to bad cold-start recommendations as it will set a very small regularization for users who have no 'X' data.

Warning: in smaller datasets, using this option can result in top-N recommendations having mostly items with very few interactions (see parameter 'scale_bias_const').

`scale_lam_sideinfo` Whether to scale (increase) the regularization parameter for each row of the "A" and "B" matrices according to the number of non-missing entries in both 'X' and the side info matrices 'U' and 'I'. If passing 'TRUE' here, 'scale_lam' will also be assumed to be 'TRUE'.

`scale_bias_const` When passing 'scale_lam=TRUE' and 'user_bias=TRUE' or 'item_bias=TRUE', whether to apply the same scaling to the regularization **of the biases** to all users and items, according to the average number of non-missing entries rather than to the number of entries for each specific user/item.

While this tends to result in worse RMSE, it tends to make the top-N recommendations less likely to select items with only a few interactions from only a few users.

Ignored when passing 'scale_lam=FALSE' or not using user/item biases.

`k_user` Number of factors in the factorizing 'A' and 'C' matrices which will be used only for the 'U' and 'U_bin' matrices, while being ignored for the 'X' matrix. These will be the first factors of the matrices once the model is fit. Will be counted in addition to those already set by 'k'.

k_item — Number of factors in the factorizing ‘B‘ and ‘D‘ matrices which will be used only for the ‘I‘ and ‘I_bin‘ matrices, while being ignored for the ‘X‘ matrix. These will be the first factors of the matrices once the model is fit. Will be counted in addition to those already set by ‘k‘.

k_main — For the ‘CMF‘ and ‘CMF_implicit‘ models, this denotes the number of factors in the factorizing ‘A‘ and ‘B‘ matrices which will be used only for the ‘X‘ matrix, while being ignored for the ‘U‘, ‘U_bin‘, ‘I‘, and ‘I_bin‘ matrices. For the ‘OMF_explicit‘ model, this denotes the number of factors which are determined without the user/item side information. These will be the last factors of the matrices once the model is fit. Will be counted in addition to those already set by ‘k‘.

w_main — Weight in the optimization objective for the errors in the factorization of the ‘X‘ matrix.

w_user — For the ‘CMF‘ and ‘CMF_implicit‘ models, this denotes the weight in the optimization objective for the errors in the factorization of the ‘U‘ and ‘U_bin‘ matrices. For the ‘OMF_explicit‘ model, this denotes the multiplier for the effect of the user attributes in the final factor matrices.

Ignored when passing neither ‘U‘ nor ‘U_bin‘.

w_item — For the ‘CMF‘ and ‘CMF_implicit‘ models, this denotes the weight in the optimization objective for the errors in the factorization of the ‘I‘ and ‘I_bin‘ matrices. For the ‘OMF_explicit‘ model, this denotes the multiplier for the effect of the item attributes in the final factor matrices.

Ignored when passing neither ‘I‘ nor ‘I_bin‘.

w_implicit — Weight in the optimization objective for the errors in the factorizations of the implicit ‘X‘ matrices. Note that, depending on the sparsity of the data, the sum of errors from these factorizations might be much larger than for the original ‘X‘ and a smaller value will perform better. It is recommended to tune this parameter carefully. Ignored when passing ‘add_implicit_features=FALSE‘.

l1_lambda — Regularization parameter to apply to the L1 norm of the model matrices. Can also pass different values for each matrix (see ‘lambda‘ for details). Note that, when adding L1 regularization, the model will be fit through a coordinate descent procedure, which is significantly slower than the Cholesky method with L2 regularization. Only supported with the ALS method.

Not recommended.

center_U — Whether to center the ’U’ matrix column-by-column. Be aware that this is a simple mean centering without regularization. One might want to turn this option off when using ‘NA_as_zero_user=TRUE‘.

center_I — Whether to center the ’I’ matrix column-by-column. Be aware that this is a simple mean centering without regularization. One might want to turn this option off when using ‘NA_as_zero_item=TRUE‘.

maxiter — Maximum L-BFGS iterations to perform. The procedure will halt if it has not converged after this number of updates. Note that the ‘CMF‘ model is likely to require fewer iterations to converge compared to other models, whereas the ‘ContentBased‘ model, which optimizes a highly non-linear function, will require more iterations and benefits from using more correction pairs. Using

higher regularization values might also decrease the number of required iterations. Pass zero for no L-BFGS iterations limit. If the procedure is spending hundreds of iterations without any significant decrease in the loss function or gradient norm, it's highly likely that the regularization is too low.

Ignored when using the ALS method.

`niter` Number of alternating least-squares iterations to perform. Note that one iteration denotes an update round for all the matrices rather than an update of a single matrix. In general, the more iterations, the better the end result. Ignored when using the L-BFGS method.

Typical values are 6 to 30.

`parallelize` How to parallelize gradient calculations when using more than one thread with 'method='lbfgs''. Passing 'separate' will iterate over the data twice - first by rows and then by columns, letting each thread calculate results for each row and column, whereas passing 'single' will iterate over the data only once, and then sum the obtained results from each thread. Passing 'separate' is much more memory-efficient and less prone to irreproducibility of random seeds, but might be slower for typical use-cases. Ignored when passing 'nthreads=1', or when using the ALS method, or when compiling without OpenMP support.

`corr_pairs` Number of correction pairs to use for the L-BFGS optimization routine. Recommended values are between 3 and 7. Note that higher values translate into higher memory requirements. Ignored when using the ALS method.

`max_cg_steps` Maximum number of conjugate gradient iterations to perform in an ALS round. Ignored when passing 'use_cg=FALSE' or using the L-BFGS method.

`precondition_cg`

Whether to use Jacobi preconditioning for the conjugate gradient procedure. In general, this type of preconditioning is not beneficial (makes the algorithm slower) as the factor variables tend to be in the same scale, but it might help when using non-shared factors.

Note that, when using preconditioning, the procedure will not check for convergence, taking instead a fixed number of steps (given by 'max_cg_steps') at each iteration regardless of whether it has reached the optimum already.

Ignored when passing 'use_cg=FALSE' or 'method="lbfgs"'.

`finalize_chol` When passing 'use_cg=TRUE' and using the ALS method, whether to perform the last iteration with the Cholesky solver. This will make it slower, but will avoid the issue of potential mismatches between the resulting factors inside the model object and calls to factors or similar with the same data.

`NA_as_zero` Whether to take missing entries in the 'X' matrix as zeros (only when the 'X' matrix is passed as a sparse matrix or as a 'data.frame') instead of ignoring them. This is a different model from the implicit-feedback version with weighted entries, and it's a much faster model to fit. Note that passing 'TRUE' will affect the results of the functions factors and factors_single (as it will assume zeros instead of missing). It is possible to obtain equivalent results to the implicit-feedback model if passing 'TRUE' here, and then passing an 'X' with all values set to one and weights corresponding to the actual values of 'X' multiplied by 'alpha', plus 1 ('W := 1 + alpha*X' to imitate the implicit-feedback model). If

passing this option, be aware that the defaults are also to perform mean centering and add user/item biases, which might be undesirable to have together with this option. For the OMF_explicit model, this option will only affect the data to which the model is fit, while being always assumed 'FALSE' for new data (e.g. when calling 'factors').

`NA_as_zero_user`

Whether to take missing entries in the 'U' matrix as zeros (only when the 'U' matrix is passed as a sparse matrix) instead of ignoring them. Note that passing 'TRUE' will affect the results of the functions factors and factors_single if no data is passed there (as it will assume zeros instead of missing). This option is always assumed 'TRUE' for the 'ContentBased', 'OMF_explicit', and 'OMF_implicit' models.

`NA_as_zero_item`

Whether to take missing entries in the 'I' matrix as zeros (only when the 'I' matrix is passed as a sparse matrix) instead of ignoring them. This option is always assumed 'TRUE' for the 'ContentBased', 'OMF_explicit', and 'OMF_implicit' models.

`nonneg`

Whether to constrain the 'A' and 'B' matrices to be non-negative. In order for this to work correctly, the 'X' input data must also be non-negative. This constraint will also be applied to the 'Ai' and 'Bi' matrices if passing 'add_implicit_features=TRUE'.

Important: be aware that the default options are to perform mean centering and to add user and item biases, which might be undesirable and hinder performance when having non-negativity constraints (especially mean centering).

This option is not available when using the L-BFGS method. Note that, when determining non-negative factors, it will always use a coordinate descent method, regardless of the value passed for 'use_cg' and 'finalize_chol'.

When used for recommender systems, one usually wants to pass 'FALSE' here.

For better results, do not use centering alongside this option, and use a higher regularization coupled with more iterations..

`nonneg_C`

Whether to constrain the 'C' matrix to be non-negative. In order for this to work correctly, the 'U' input data must also be non-negative.

Note: by default, the 'U' data will be centered by columns, which doesn't play well with non-negativity constraints. One will likely want to pass 'center_U=FALSE' along with this.

`nonneg_D`

Whether to constrain the 'D' matrix to be non-negative. In order for this to work correctly, the 'I' input data must also be non-negative.

Note: by default, the 'I' data will be centered by columns, which doesn't play well with non-negativity constraints. One will likely want to pass 'center_I=FALSE' along with this.

`max_cd_steps`

Maximum number of coordinate descent updates to perform per iteration. Pass zero for no limit. The procedure will only use coordinate descent updates when having L1 regularization and/or non-negativity constraints. This number should usually be larger than 'k'.

`precompute_for_predictions`

Whether to precompute some of the matrices that are used when making predictions from the model. If 'FALSE', it will take longer to generate predictions

or top-N lists, but will use less memory and will be faster to fit the model. If passing 'FALSE', can be recomputed later on-demand through function precompute.for.predictions.

Note that for 'ContentBased', 'OMF_explicit', and 'OMF_implicit', this parameter will always be assumed to be 'TRUE', due to requiring the original matrices for the pre-computations.

`include_all_X` When passing an input 'X' which has less columns than rows in 'I', whether to still make calculations about the items which are in 'I' but not in 'X'. This has three effects: (a) the topN functionality may recommend such items, (b) the precomptued matrices will be less usable as they will include all such items, (c) it will be possible to pass 'X' data to the new factors or topN functions that include such columns (rows of 'I'). This option is ignored when using 'NA_as_zero', and is only relevant for the 'CMF' model as all the other models will have the equivalent of 'TRUE' here.

`verbose` Whether to print informational messages about the optimization routine used to fit the model. Be aware that, if passing 'FALSE' and using the L-BFGS method, the optimization routine will not respond to interrupt signals.

`print_every` Print L-BFGS convergence messages every n-iterations. Ignored when not using the L-BFGS method.

`handle_interrupt`

When receiving an interrupt signal, whether the model should stop early and leave a usable object with the parameters obtained up to the point when it was interrupted (when passing 'TRUE'), or raise an interrupt exception without producing a fitted model object (when passing 'FALSE').

`seed` Seed to use for random number generation. If passing 'NULL', will draw a non-reproducible random integer to use as seed.

`nthreads` Number of parallel threads to use. Note that, the more threads that are used, the higher the memory consumption.

`alpha` Weighting parameter for the non-zero entries in the implicit-feedback model. See [3] for details. Note that, while the author's suggestion for this value is 40, other software such as the Python package 'implicit' use a value of 1, whereas Spark uses a value of 0.01 by default, and values higher than 10 are unlikely to improve results. If the data has very high values, might even be beneficial to put a very low value here - for example, for the LastFM-360K, values below 1 might give better results.

`apply_log_transf`

Whether to apply a logarithm transformation on the values of 'X' (i.e. 'X := log(X)')

`implicit` (Only selectable for the 'MostPopular' model) Whether to use the implicit-feedback model, in which the 'X' matrix is assumed to have only binary entries and each of them having a weight in the loss function given by the observer user-item interactions and other parameters.

`add_intercepts` (Only for 'ContentBased', 'OMF_explicit', 'OMF_implicit') Whether to add intercepts/biases to the user/item attribute matrices.

`start_with_ALS` (Only for 'ContentBased') Whether to determine the initial coefficients through an ALS procedure. This might help to speed up the procedure by starting closer

to an optimum. This option is not available when the side information is passed as sparse matrices.

Note that this option will not work (will throw an error) if there are users or items without side information, or if the input data is otherwise problematic (e.g. users/items which are duplicates of each other).

k_sec (Only for 'OMF_explicit') Number of factors in the factorizing matrices which are determined exclusively from user/item attributes. These will be at the beginning of the 'C' and 'D' matrices once the model is fit. If there are no attributes for a given matrix (user/item), then that matrix will have an extra 'k_sec' factors (e.g. if passing user side info but not item side info, then the 'B' matrix will have an extra 'k_sec' factors). Will be counted in addition to those already set by 'k'. Not supported when using 'method='als''.

For a different model having only 'k_sec' with 'k=0' and 'k_main=0', see the 'ContentBased' model

Details

In more details, the models predict the values of 'X' as follows:

- 'CMF': $\mathbf{X} \approx \mathbf{A}\mathbf{B}^T + \mu + \mathbf{b}_u + \mathbf{b}_i$, where μ is the global mean for the non-missing entries in 'X', and $\mathbf{b}_u, \mathbf{b}_i$ are the user and item biases (column and row vector, respectively). In addition, the other matrices are predicted as $\mathbf{U} \approx \mathbf{A}\mathbf{C}^T + \mu_U$ and $\mathbf{I} \approx \mathbf{B}\mathbf{D}^T + \mu_I$, where μ_U, μ_I are the column means from the side info matrices, which are determined as a simple average with no regularization (these are row vectors), and if having binary variables, also $\mathbf{U}_{bin} \approx \sigma(\mathbf{A}\mathbf{C}_{bin}^T)$ and $\mathbf{I}_{bin} \approx \sigma(\mathbf{B}\mathbf{D}_{bin}^T)$, where σ is a sigmoid function ($\sigma(x) = \frac{1}{1+e^{-x}}$). Under the options 'NA_as_zero_*', the mean(s) for that matrix are not added into the model for simplicity. For the implicit features option, the other matrices are predicted simply as $\mathbf{I}_x \approx \mathbf{A}\mathbf{B}_i$, $\mathbf{I}_x^T \approx \mathbf{B}\mathbf{A}_i$.

  If using 'k_user', 'k_item', 'k_main', then for 'X', only columns '1' through 'k+k_user' are used in the approximation of 'U', and only columns 'k_user+1' through 'k_user+k+k_main' are used for the approximation of 'X' (similar thing for 'B' with 'k_item'). The implicit factors matrices ($\mathbf{A}_i, \mathbf{B}_i$) always use the same components/factors as 'X'.

  Be aware that the functions for determining new factors will by default omit the bias term in the output.

- 'CMF_implicit': $\mathbf{X} \approx \mathbf{A}\mathbf{B}^T$, while 'U' and 'I' remain the same as for 'CMF', and the ordering of the non-shared factors is the same. Note that there is no mean centering or user/item biases in the implicit-feedback model, but if desired, the 'CMF' model can be made to mimic 'CMF_implicit' while still accommodating for mean centering and biases.

- 'MostPopular': $\mathbf{X} \approx \mu + \mathbf{b}_u + \mathbf{b}_i$ (when using 'implicit=FALSE') or $\mathbf{X} \approx \mathbf{b}_i$ (when using 'implicit=TRUE').

- 'ContentBased': $\mathbf{X} \approx \mathbf{A}_m\mathbf{B}_m^T$, where $\mathbf{A}_m = \mathbf{U}\mathbf{C} + \mathbf{b}_C$ and $\mathbf{B}_m = \mathbf{I}\mathbf{D} + \mathbf{b}_D$ - the $\mathbf{b}_C, \mathbf{b}_D$ are per-column/factor intercepts (these are row vectors).

- 'OMF_explicit': $\approx \mathbf{A}_m\mathbf{B}_m^T + \mu + \mathbf{b}_u + \mathbf{b}_i$, where $\mathbf{A}_m = w_u(\mathbf{U}\mathbf{C} + \mathbf{b}_C) + \mathbf{A}$ and $\mathbf{B}_m = w_i(\mathbf{I}\mathbf{D} + \mathbf{b}_D) + \mathbf{B}$. If passing 'k_sec' and/or 'k_main', then columns '1' through 'k_sec' of $\mathbf{A}_m, \mathbf{B}_m$ are determined as those same columns from $\mathbf{A}, \mathbf{B}$, while $\mathbf{U}\mathbf{C} + \mathbf{b}_C, \mathbf{I}\mathbf{D} + \mathbf{b}_D$ will be shorter by 'k_sec' columns (alternatively, can be though of as having those columns artificially set to zeros), and columns 'k_sec+k+1' through 'k_sec+k+k_main' of $\mathbf{A}_m, \mathbf{B}_m$ are determined as those last 'k_main' columns of $\mathbf{U}\mathbf{C}+\mathbf{b}_C, \mathbf{I}\mathbf{D}+\mathbf{b}_D$, while $\mathbf{A}, \mathbf{B}$ will be shorter

by ‘k_main‘ columns (alternatively, can be though of as having those columns artificially set to zeros). If one of $\mathbf{U}$ or $\mathbf{I}$ is missing, then the corresponding $\mathbf{A}$ or $\mathbf{B}$ matrix will be extended by ‘k_sec‘ columns (which will not be zeros) and the corresponding prediction matrix ($\mathbf{A}_m, \mathbf{B}_m$) will be equivalent to that matrix (which was the free offset in the presence of side information).

- ‘OMF_implicit‘: $\mathbf{X} \approx \mathbf{A}_m \mathbf{B}_m^T$, with $\mathbf{A}_m, \mathbf{B}_m$ remaining the same as for ‘OMF_explicit‘.

When calling the prediction functions, new data is always transposed or deep copied before passing them to the underlying C functions - as such, for the ‘ContentBased‘ model, it might be faster to use the matrices directly instead (all these matrices will be under ‘model$matrices‘, but will be transposed).

The precomputed matrices, when they are square, will only contain the lower triangle only, as they are symmetric. For ‘CMF‘ and ‘CMF_implicit‘, one might also see variations of a new matrix called ‘Be‘ (extended ‘B‘ matrix), which is from reference [1] and defined as $\mathbf{B}_e = [[\mathbf{0}, \mathbf{B}_s, \mathbf{B}_m], [\mathbf{C}_a, \mathbf{C}_s, \mathbf{0}]]$, where $\mathbf{B}_s$ are columns ‘k_item+1‘ through ‘k_item+k‘ from ‘B‘, $\mathbf{B}_m$ are columns ‘k_item+k+1‘ through ‘k_item+k+k_main‘ from ‘B‘, $\mathbf{C}_a$ are columns ‘1‘ through ‘k_user‘ from ‘C‘, and $\mathbf{C}_s$ are columns ‘k_user+1‘ through ‘k_user+k‘ from ‘C‘. This matrix is used for the closed-form solution of a given vector of ‘A‘ in the functions for predicting on new data (see reference [1] for details or if you would like to use your own solver with the fitted matrices from this package), as long as there are no binary columns to which to apply a transformation, in which case it will always solve them with the L-BFGS method.

When using user biases, the precomputed matrices will have an extra column, which is derived by adding an extra column to ‘B‘ (at the end) consisting of all ones (this is how the user biases are calculated).

For the implicit-feedback models, the weights of the positive entries (defined as the non-missing entries in ‘X‘) will be given by $W = 1 + \alpha \mathbf{X}$.

For the ‘OMF‘ models, the ‘ALS‘ method will first find a solution for the equivalent ‘CMF‘ problem with no side information, and will then try to predict the resulting matrices given the user/item attributes, assigning the residuals as the free offsets. While this might sound reasonable, in practice it tends to give rather different results than when fit through the L-BFGS method. Strictly speaking, the regularization parameter in this case is applied to the $\mathbf{A}_m, \mathbf{B}_m$ matrices, and the prediction functions for new data will offer an option ‘exact‘ for determining whether to apply the regularization to the $\mathbf{A}, \mathbf{B}$ matrices instead.

For reproducibility, the initializations of the model matrices (always initialized as ‘~ Normal(0, 1)‘ for ‘CMF‘ and as ‘~ Uniform(0,1)‘ for ‘CMF_implicit‘) can be controlled through the seed parameter, but if using parallelizations, there are potential sources of irreproducibility of random seeds due to parallelized aggregations and/or BLAS function calls, which is especially problematic for the L-BFGS method with ‘parallelize='single'‘.

In order to further avoid potential decimal differences in the factors obtained when fitting the model and when calling the prediction functions on new data, when the data is sparse, it's necessary to sort it beforehand by columns/items and also pass the data data with item indices sorted beforehand to the prediction functions. The package does not perform any indices sorting or de-duplication of entries of sparse matrices.

Value

Returns a model object (class named just like the function that produced it, plus general class ‘cmfrec‘) on which methods such as topN and factors can be called. The returned object will have

the following fields:

- ‘info‘: will contain the hyperparameters, problem dimensions, and other information such as the number of threads, as passed to the function that produced the model. The number of threads (‘nthreads‘) might be modified after-the-fact. If ‘X‘ is a ‘data.frame‘, will also contain the re-indexing of users and items under ‘user_mapping‘ and ‘item_mapping‘, respectively. For the L-BFGS method, will also contain the number of function evaluations (‘nfev‘) and number of updates (‘nupd‘) that were performed.
- ‘matrices‘: will contain the fitted model matrices (see section ‘Description‘ for the naming and for details on what they represent), but note that they will be transposed (due to R’s column-major representation of matrices) and it is recommended to use the package’s prediction functionality instead of taking the matrices directly.
- ‘precomputed‘: will contain some pre-computed calculations based on the model matrices which might help speed up predictions on new data.

Evaluating models

Metrics for implicit-feedback recommendations or model quality can be calculated using the recometrics package.

Performance tips

It is recommended to have the RhpcBLASctl package installed for better performance - if available, will be used to control the number of internal BLAS threads before entering a multi-threaded region, in order to avoid oversubscription of threads. This can become an issue when using OpenBLAS if it is the ’pthreads’ variant.

This package relies heavily on BLAS and LAPACK functions, and benefits from SIMD vectorization. For better performance, it is recommended to use an optimized backed such as MKL or OpenBLAS, and to enable additional compiler optimizations that are not used by default in R.

See this guide for details: installing optimized libraries.

References

1. Cortes, David. "Cold-start recommendations in Collective Matrix Factorization." arXiv preprint arXiv:1809.00366 (2018).
2. Singh, Ajit P., and Geoffrey J. Gordon. "Relational learning via collective matrix factorization." Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining. 2008.
3. Hu, Yifan, Yehuda Koren, and Chris Volinsky. "Collaborative filtering for implicit feedback datasets." 2008 Eighth IEEE International Conference on Data Mining. Ieee, 2008.
4. Takacs, Gabor, Istvan Pilaszy, and Domonkos Tikk. "Applications of the conjugate gradient method for implicit feedback collaborative filtering." Proceedings of the fifth ACM conference on Recommender systems. 2011.
5. Rendle, Steffen, Li Zhang, and Yehuda Koren. "On the difficulty of evaluating baselines: A study on recommender systems." arXiv preprint arXiv:1905.01395 (2019).

6. 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.
7. 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

```
### See the package vignette for an extended version of this example

library(cmfrec)
if (require("recommenderlab") && require("MatrixExtra")) {
    ### Load the ML100K dataset (movie ratings)
    ### (users are rows, items are columns)
    data("MovieLense")
    X <- as.coo.matrix(MovieLense@data)

    ### Will add basic side information about the users
    U <- MovieLenseUser
    U$id      <- NULL
    U$zipcode <- NULL
    U <- model.matrix(~.-1, data=U)

    ### Will additionally use the item genres as side info
    I <- MovieLenseMeta
    I$title <- NULL
    I$year  <- NULL
    I$url   <- NULL
    I <- as.coo.matrix(I)

    ### Fit a factorization model
    ### (it's recommended to change the hyperparameters
    ###  and use multiple threads)
    model <- CMF(X=X, U=U, I=I, k=10L, niter=5L,
                 NA_as_zero_item=TRUE,
                 verbose=FALSE, nthreads=1L)

    ### Predict rating for entries X[1,3], X[2,5], X[10,9]
    ### (first ID is the user, second is the movie)
    predict(model, user=c(1,2,10), item=c(3,5,9))

    ### Recommend top-5 for user ID = 10
    ### (Note that 'MatrixExtra' makes this return a 'sparseVector')
    seen_by_user <- MovieLense@data[10, , drop=TRUE]@i
    rec <- topN(model, user=10, n=5, exclude=seen_by_user)
    rec

    ### Print them in a more understandable format
    movie_names <- colnames(X)
    n_ratings <- colSums(as.csc.matrix(X, binary=TRUE))
    avg_ratings <- colSums(as.csc.matrix(X)) / n_ratings
```

```
    print_recommended <- function(rec, txt) {
        cat(txt, ":\n",
            paste(paste(1:length(rec), ". ", sep=""),
                  movie_names[rec],
                  " - Avg rating:", round(avg_ratings[rec], 2),
                  ", #ratings: ", n_ratings[rec],
                  collapse="\n", sep=""),
            "\n", sep="")
    }
    print_recommended(rec, "Recommended for user_id=10")


    ### Recommend assuming it is a new user,
    ### based on its data (ratings + side info)
    x_user <- X[10, , drop=TRUE] ## <- this is a 'sparseVector'
    u_user <- U[10, ]
    rec_new <- topN_new(model, n=5, X=x_user, U=u_user, exclude=seen_by_user)
    cat("lists are identical: ", identical(rec_new, rec), "\n")

    ### Recommend based on side information alone
    ### (a.k.a. cold-start recommendation)
    rec_cold <- topN_new(model, n=5, U=u_user)
    print_recommended(rec_cold, "Recommended based on side info")

    ### Obtain factors for the user
    factors_user <- model$matrices$A[, 10, drop=TRUE]

    ### Re-calculate them based on the data
    factors_new <- factors_single(model, X=x_user, U=u_user)

    ### Should be very close, but due to numerical precision,
    ### might not be exactly equal (see section 'Details')
    cat("diff: ", factors_user - factors_new, "\n")

    ### Can also calculate them in batch
    ### (slicing is provided by package "MatrixExtra")
    Xslice <- as.csr.matrix(X)[1:10, , drop=FALSE]
    Uslice <- U[1:10, , drop=FALSE]
    factors_multiple <- factors(model, X=Xslice, U=Uslice)
    cat("diff: ", factors_multiple[10, , drop=TRUE] - factors_new, "\n")

    ### Can make cold-start predictions, e.g.
    ### predict how would users [1,2,3] rate a new item,
    ### given it's side information (here it's item ID = 5)
    predict_new_items(model, user=c(1,2,3), item=c(1,1,1), I=I[5, ])
}
```

---

imputeX *Impute missing entries in 'X' data*

Description

Replace ‘NA‘/‘NaN‘ values in new ‘X‘ data according to the model predictions, given that same ‘X‘ data and optionally ‘U‘ data.

Note: this function will not perform any internal re-indexing for the data. If the ‘X‘ to which the data was fit was a ‘data.frame‘, the numeration of the items will be under ‘model$info$item_mapping‘. There is also a function predict_new which will let the model do the appropriate reindexing.

Usage

```
imputeX(
  model,
  X,
  weight = NULL,
  U = NULL,
  U_bin = NULL,
  nthreads = model$info$nthreads
)
```

Arguments

model	A collective matrix factorization model as output by function CMF. This functionality is not available for the other model classes.
X	New ‘X‘ data with missing values which will be imputed. Must be passed as a dense matrix from base R (class ‘matrix‘).
weight	Associated observation weights for entries in ‘X‘. If passed, must have the same shape as ‘X‘.
U	New ‘U‘ data, with rows matching to those of ‘X‘. Can be passed in the following formats: • A sparse COO/triplets matrix, either from package ‘Matrix‘ (class ‘dgTMatrix‘), or from package ‘SparseM‘ (class ‘matrix.coo‘). • A sparse matrix in CSR format, either from package ‘Matrix‘ (class ‘dgRMatrix‘), or from package ‘SparseM‘ (class ‘matrix.csr‘). Passing the input as CSR is faster than COO as it will be converted internally. • A sparse row vector from package ‘Matrix‘ (class ‘dsparseVector‘). • A dense matrix from base R (class ‘matrix‘), with missing entries set as ‘NA‘/‘NaN‘. • A dense row vector from base R (class ‘numeric‘). • A ‘data.frame‘.
U_bin	New binary columns of ‘U‘ (rows matching to those of ‘X‘). Must be passed as a dense matrix from base R or as a ‘data.frame‘.
nthreads	Number of parallel threads to use.

Details

If using the matrix factorization model as a general missing-value imputer, it’s recommended to:

- Fit a model without user biases.
- Set a lower regularization for the item biases than for the matrices.
- Tune the regularization parameter(s) very well.

In general, matrix factorization works better for imputation of selected entries of sparse-and-wide matrices, whereas for dense matrices, the method is unlikely to provide better results than mean/median imputation, but it is nevertheless provided for experimentation purposes.

Value

The ‘X‘ matrix with its missing values imputed according to the model predictions.

Examples

```
library(cmfrec)

### Simplest example
SeqMat <- matrix(1:50, nrow=10)
SeqMat[2,1] <- NaN
SeqMat[8,3] <- NaN
m <- CMF(SeqMat, k=1, lambda=1e-10, nthreads=1L, verbose=FALSE)
imputeX(m, SeqMat)


### Better example with multivariate normal data
if (require("MASS")) {
    ### Generate random data, set some values as NA
    set.seed(1)
    n_rows <- 1000
    n_cols <- 5
    mu <- rnorm(n_cols)
    S <- matrix(rnorm(n_cols^2), nrow = n_cols)
    S <- t(S) %*% S
    X <- MASS::mvrnorm(n_rows, mu, S)
    X_na <- X
    values_NA <- matrix(runif(n_rows*n_cols) < .15, nrow=n_rows)
    X_na[values_NA] <- NaN

    ### In the event that any column is fully missing
    if (any(colSums(is.na(X_na)) == n_rows)) {
        cols_remove <- colSums(is.na(X_na)) == n_rows
        X_na <- X_na[, !cols_remove, drop=FALSE]
        values_NA <- values_NA[, !cols_remove, drop=FALSE]
    }

    ### Impute missing values with model
    model <- CMF(X_na, k=3, lambda=c(0,0,1,1,1,1),
                 user_bias=FALSE,
                 verbose=FALSE, nthreads=1L)
    X_imputed <- imputeX(model, X_na)
    cat(sprintf("RMSE for imputed values w/model: %f\n",
                sqrt(mean((X[values_NA] - X_imputed[values_NA])^2))))
```

```
    ### Compare against simple mean imputation
    X_means <- apply(X_na, 2, mean, na.rm=TRUE)
    X_imp_mean <- X_na
    for (cl in 1:n_cols)
        X_imp_mean[values_NA[,cl], cl] <- X_means[cl]
    cat(sprintf("RMSE for imputed values w/means: %f\n",
                sqrt(mean((X[values_NA] - X_imp_mean[values_NA])^2))))
}
```

`item_factors` *Determine latent factors for a new item*

Description

Calculate latent factors for a new item, based on either new ‘X‘ data, new ‘I‘ data, or both.

Be aware that the package is user/row centric, and this function is provided for quick experimentation purposes only. Calculating item factors will be slower than calculating user factors (except for the ‘ContentBased‘ model for which both types of predictions are equally fast and equally supported). as it will not make usage of the precomputed matrices. If item-based predictions are required, it’s recommended to use instead the function swap.users.and.items and then use the resulting object with factors_single or factors.

Usage

```
item_factors(
  model,
  X = NULL,
  X_col = NULL,
  X_val = NULL,
  I = NULL,
  I_col = NULL,
  I_val = NULL,
  I_bin = NULL,
  weight = NULL,
  output_bias = FALSE
)
```

Arguments

`model`	A collective matrix factorization model from this package - see fit_models for details.
`X`	New ‘X‘ data, either as a numeric vector (class ‘numeric‘), or as a sparse vector from package ‘Matrix‘ (class ‘dsparseVector‘). If the ‘X‘ to which the model was fit was a ‘data.frame‘, the user/row indices will have been reindexed internally, and the numeration can be found under ‘model$info$user_mapping‘. Alternatively, can instead pass the column indices and values and let the model

reindex them (see ‘X_col‘ and ‘X_val‘). Should pass at most one of ‘X‘ or ‘X_col‘+‘X_val‘.
Be aware that, unlikely in pretty much every other function in this package, here the values are for one **column** of ‘X‘, not one **row** like in e.g. factors_single.
Dense ‘X‘ data is not supported for ‘CMF_implicit‘ or ‘OMF_implicit‘.
Not supported for the ‘ContentBased‘ model.

`X_col` New ‘X‘ data in sparse vector format, with ‘X_col‘ denoting the users/rows which are not missing. If the ‘X‘ to which the model was fit was a ‘data.frame‘, here should pass IDs matching to the first column of that ‘X‘, which will be reindexed internally. Otherwise, should have **row** indices with numeration starting at 1 (passed as an integer vector). Should pass at most one of ‘X‘ or ‘X_col‘+‘X_val‘.
Not supported for the ‘ContentBased‘ model.

`X_val` New ‘X‘ data in sparse vector format, with ‘X_val‘ denoting the associated values to each entry in ‘X_col‘ (should be a numeric vector of the same length as ‘X_col‘). Should pass at most one of ‘X‘ or ‘X_col‘+‘X_val‘.
Not supported for the ‘ContentBased‘ model.

`I` New ‘I‘ data, either as a numeric vector (class ‘numeric‘), or as a sparse vector from package ‘Matrix‘ (class ‘dsparseVector‘). Alternatively, if ‘I‘ is sparse, can instead pass the indices of the non-missing columns and their values separately (see ‘I_col‘). Should pass at most one of ‘I‘ or ‘I_col‘+‘I_val‘.

`I_col` New ‘I‘ data in sparse vector format, with ‘I_col‘ denoting the attributes/columns which are not missing. Should have numeration starting at 1 (should be an integer vector). Should pass at most one of ‘I‘ or ‘I_col‘+‘I_val‘.

`I_val` New ‘I‘ data in sparse vector format, with ‘I_val‘ denoting the associated values to each entry in ‘I_col‘ (should be a numeric vector of the same length as ‘I_col‘). Should pass at most one of ‘I‘ or ‘I_col‘+‘I_val‘.

`I_bin` Binary columns of ‘I‘ on which a sigmoid transformation will be applied. Should be passed as a numeric vector. Note that ‘I‘ and ‘I_bin‘ are not mutually exclusive.

`weight` (Only for the explicit-feedback models) Associated weight to each non-missing observation in ‘X‘. Must have the same number of entries as ‘X‘ - that is, if passing a dense vector of length ‘m‘, ‘weight‘ should be a numeric vector of length ‘m‘ too, if passing a sparse vector, should have a length corresponding to the number of non-missing elements.

`output_bias` Whether to also return the item bias determined by the model given the data in ‘X‘ (for explicit-feedback models fit with item biases).

Value

If passing ‘output_bias=FALSE‘, will return a vector with the obtained latent factors for this item. If passing ‘output_bias=TRUE‘, the result will be a list with entry ‘factors‘ having the above vector, and entry ‘bias‘ having the estimated bias.

See Also

factors_single predict_new_items

precompute.for.predictions *Precompute matrices to use for predictions*

Description

Pre-computes internal matrices which might be used to speed up computations on new data in the CMF and CMF_implicit models. This function does not need to be called when passing 'precompute_for_predictions=TRUE'.

Usage

```
precompute.for.predictions(model)
```

Arguments

model	A collective matrix factorization model object, for which the pre-computed matrices will be calculated.

Value

The same model object, with the pre-calculated matrices inside it.

predict.cmfrec *Predict entries in the factorized 'X' matrix*

Description

Predict entries in the 'X' matrix according to the model at the combinations [row,column] given by the entries in 'user' and 'item' (e.g. passing 'user=c(1,2,3), item=c(1,1,1)' will predict X[1,1], X[2,1], X[3,1]).

Alternatively, might pass a sparse matrix, in which case it will make predictions for all of its non-missing entries.

Invalid combinations (e.g. rows and columns outside of the range of 'X' to which the model was fit) will be filled with global mean plus biases if applicable for 'CMF_explicit', and with NAs for the other models.

For example usage, see the main section fit_models.

Usage

```
## S3 method for class 'cmfrec'
predict(object, user, item = NULL, nthreads = object$info$nthreads, ...)
```

Arguments

`object`	A collective matrix factorization model from this package - see fit_models for details.
`user`	The user IDs for which to make predictions. If ‘X‘ to which the model was fit was a ‘data.frame‘, should pass IDs matching to the first column of ‘X‘ (the user indices, should be a character vector), otherwise should pass row numbers for ‘X‘, with numeration starting at 1 (should be an integer vector). If passing a single entry for ‘user‘ and ‘item‘ has more entries, will predict all the entries in ‘item‘ for that single ‘user.‘ Alternatively, might instead pass a sparse matrix in COO/triplets formats, for which the **non-missing** entries will be predicted, in which case it its not necessary to pass ‘item‘. If passing a sparse matrix, can be from package ‘Matrix‘ (class ‘dgTMatrix‘ or ‘ngTMatrix‘) or from package ‘SparseM‘ (class ‘matrix.coo‘). If using the package ‘softImpute‘, its objects of class ‘incomplete‘ might be convertable to ‘Matrix‘ objects through e.g. ‘as(as(X, "TsparseMatrix"), "nMatrix")‘.
`item`	The item IDs for which to make predictions - see the documentation about ‘user‘ for details about the indexing. If passing a single entry for ‘item‘ and ‘user‘ has more entries, will predict all the entries in ‘user‘ for that single ‘item‘. If passing a sparse matrix as ‘user‘, ‘item‘ will be ignored.
`nthreads`	Number of parallel threads to use.
`...`	Not used.

Value

A numeric vector with the predicted values at the requested combinations. If the ‘user‘ passed was a sparse matrix, and it was not of class ‘ngTMatrix‘, will instead return a sparse matrix of the same format, with the non-missing entries set to the predicted values.

See Also

predict_new topN

`predict_new` *Predict entries in new ‘X‘ data*

Description

Predict entries in columns of the ‘X‘ matrix for new users/rows given their new ‘X‘ and/or ‘U‘ data at the combinations [row,column] given by the entries in ‘user‘ and ‘item‘ (e.g. passing ‘user=c(1,2,3), item=c(1,1,1)‘ will predict X[1,1], X[2,1], X[3,1]).

Note: this function will not perform any internal re-indexing for the data. If the ‘X‘ to which the data was fit was a ‘data.frame‘, the numeration of the items will be under ‘model$info$item_mapping‘.

Usage

```
predict_new(model, ...)

## S3 method for class 'CMF'
predict_new(
  model,
  items,
  rows = NULL,
  X = NULL,
  U = NULL,
  U_bin = NULL,
  weight = NULL,
  nthreads = model$info$nthreads,
  ...
)

## S3 method for class 'CMF_implicit'
predict_new(
  model,
  items,
  rows = NULL,
  X = NULL,
  U = NULL,
  nthreads = model$info$nthreads,
  ...
)

## S3 method for class 'OMF_explicit'
predict_new(
  model,
  items,
  rows = NULL,
  X = NULL,
  U = NULL,
  weight = NULL,
  exact = FALSE,
  nthreads = model$info$nthreads,
  ...
)

## S3 method for class 'OMF_implicit'
predict_new(
  model,
  items,
  rows = NULL,
  X = NULL,
  U = NULL,
  nthreads = model$info$nthreads,
```

```
  ...
)

## S3 method for class 'ContentBased'
predict_new(
  model,
  items = NULL,
  rows = NULL,
  U = NULL,
  I = NULL,
  nthreads = model$info$nthreads,
  ...
)
```

Arguments

model	A collective matrix factorization model from this package - see fit_models for details.
...	Not used.
items	The item IDs for which to make predictions. If 'X' to which the model was fit was a 'data.frame', should pass IDs matching to the second column of 'X' (the item indices, should be a character vector), otherwise should pass column numbers for 'X', with numeration starting at 1 (should be an integer vector). If passing 'I', will instead take these indices as row numbers for 'I' (only available for the ContentBased model).
rows	Rows of the new 'X'/'U' passed here for which to make predictions, with numeration starting at 1 (should be an integer vector). If not passed and there is only 1 row of data, will predict the entries in 'items' for that same row. If not passed and there is more than 1 row of data, the number of rows in the data should match with the number of entries in 'items', and will make predictions for each such combination of <entry in item, row in the data>.
X	New 'X' data, with rows denoting new users. Can be passed in the following formats: • A sparse COO/triplets matrix, either from package 'Matrix' (class 'dgTMatrix'), or from package 'SparseM' (class 'matrix.coo'). • A sparse matrix in CSR format, either from package 'Matrix' (class 'dgRMatrix'), or from package 'SparseM' (class 'matrix.csr'). Passing the input as CSR is faster than COO as it will be converted internally. • A sparse row vector from package 'Matrix' (class 'dsparseVector'). • A dense matrix from base R (class 'matrix'), with missing entries set as 'NA'/'NaN'. • A dense row vector from base R (class 'numeric'), with missing entries set as 'NA'/'NaN'. Dense 'X' data is not supported for 'CMF_implicit' or 'OMF_implicit'.
U	New 'U' data, with rows denoting new users. Can be passed in the same formats as 'X', or additionally as a 'data.frame', which will be internally converted to a matrix.

U_bin	New binary columns of 'U'. Must be passed as a dense matrix from base R or as a 'data.frame'.
weight	Associated observation weights for entries in 'X'. If passed, must have the same shape as 'X' - that is, if 'X' is a sparse matrix, should be a numeric vector with length equal to the non-missing elements, if 'X' is a dense matrix, should also be a dense matrix with the same number of rows and columns.
nthreads	Number of parallel threads to use.
exact	(In the 'OMF_explicit' model) Whether to calculate 'A' and 'Am' with the regularization applied to 'A' instead of to 'Am' (if using the L-BFGS method, this is how the model was fit). This is usually a slower procedure.
I	(For the 'ContentBased' model only) New 'I' data for which to make predictions. Supports the same formats as 'U'.

Value

A numeric vector with the predicted values for the requested combinations of users (rows in the new data) and items (columns in the old data, unless passing 'I' in which case will be rows of 'I'). Invalid combinations will be filled with NAs.

See Also

predict.cmfrec

---

predict_new_items *Predict new columns of 'X' given item attributes*

Description

Calculate the predicted values for new columns of 'X' (which were not present in the 'X' to which the model was fit) given new 'X' and/or 'I' data.

This function can predict combinations in 3 ways:

- If passing vectors for 'user' and 'item', will predict the combinations of user/item given in those arrays (e.g. if 'I' has 3 rows, and passing 'user=c(1,1,2), item=c(1,2,3)', will predict entries X[1,1], X[1,2], X[2,3], with columns of 'X' (rows of 't(X)') corresponding to the rows of 'I' passed here and users corresponding to the ones to which the model was fit).
- If passing a vector for 'user' but not for 'item', will predict the value that each user would give to the corresponding row of 'I'/'t(X)' (in this case, the number of entries in 'user' should be the same as the number of rows in 'I'/'t(X)').
- If passing a single value for 'user', will calculate all predictions for that user for the rows of 'I'/'t(X)' given in 'item', or for all rows of 'I'/'t(X)' if 'item' is not given.

Be aware that the package is user/row centric, and this function is provided for quick experimentation purposes only. Calculating item factors will be slower than calculating user factors as it will not make usage of the precomputed matrices (except for the 'ContentBased' model for which both types of predictions are equally fast and equally supported). If item-based predictions are required, it's recommended to use instead the function swap.users.and.items and then use the resulting object with predict_new.

Usage

```
predict_new_items(
  model,
  user,
  item = NULL,
  transX = NULL,
  weight = NULL,
  I = NULL,
  I_bin = NULL,
  nthreads = model$info$nthreads
)
```

Arguments

model	A collective matrix factorization model from this package - see fit_models for details.
user	User(s) for which the new entries will be predicted. If passing a single ID, will calculate all the values in 'item', or all the values in 'I'/'t(X)' (see section 'Description' for details). If the 'X' to which the model was fit was a 'data.frame', the IDs here should match with the IDs of that 'X' (its first column). Otherwise, should match with the rows of 'X' (the one to which the model was fit) with numeration starting at 1 (should be an integer vector).
item	Rows of 'I'/'transX' (unseen columns of a new 'X') for which to make predictions, with numeration starting at 1 (should be an integer vector). See 'Description' for details.
transX	New 'X' data for the items, transposed so that items denote rows and columns correspond to old users (which were in the 'X' to which the model was fit). Note that the function will not do any reindexing - if the 'X' to which the model was fit was a 'data.frame', the user numeration can be found under 'model$info$user_mapping'. Can be passed in the following formats: • A sparse COO/triplets matrix, either from package 'Matrix' (class 'dgTMatrix'), or from package 'SparseM' (class 'matrix.coo'). • A sparse matrix in CSR format, either from package 'Matrix' (class 'dgRMatrix'), or from package 'SparseM' (class 'matrix.csr'). Passing the input as CSR is faster than COO as it will be converted internally. • A sparse row vector from package 'Matrix' (class 'dsparseVector'). • A dense matrix from base R (class 'matrix'), with missing entries set as NA. • A dense vector from base R (class 'numeric'). • A 'data.frame'.
weight	Associated observation weights for entries in 'transX'. If passed, must have the same shape as 'transX' - that is, if 'transX' is a sparse matrix, should be a numeric vector with length equal to the non-missing elements, if 'transX' is a dense matrix, should also be a dense matrix with the same number of rows and columns.

I	New ‘I‘ data, with rows denoting new columns of the ‘X‘ matrix (the one to which the model was fit) and/or rows of ‘transX‘. Can be passed in the same formats as ‘transX‘, or additionally as a ‘data.frame‘.
I_bin	New binary columns of ‘I‘. Must be passed as a dense matrix from base R or as a ‘data.frame‘.
nthreads	Number of parallel threads to use.

Value

A numeric vector with the predicted value for each requested combination of (user, item). Invalid combinations will be filled with NAs.

See Also

item_factors predict.cmfrec predict_new

`print.cmfrec` *Get information about factorization model*

Description

Print basic properties of a ‘cmfrec‘ object (a base class encompassing all the models in this package).

Usage

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

Arguments

x	An object of class ‘cmfrec‘ as returned by functions CMF, CMF_implicit, MostPopular, ContentBased, OMF_explicit, OMF_implicit.
...	Extra arguments (not used).

Value

Returns the same model object that was passes as input.

summary.cmfrec *Get information about factorization model*

Description

Print basic properties of a ‘cmfrec‘ object (a base class encompassing all the models in this package). Same as the ‘print.cmfrec‘ function.

Usage

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

Arguments

object	An object of class ‘cmfrec‘ as returned by functions CMF, CMF_implicit, MostPopular, ContentBased, OMF_explicit, OMF_implicit.
...	Extra arguments (not used).

Value

No return value (information is printed).

See Also

print.cmfrec

swap.users.and.items *Swap users and items in the model*

Description

This function will swap the users and items in a given matrix factorization model. Since the package functionality is user-centric, it is generally not possible or not efficient to make predictions about items (e.g. recommend users for a given item or calculate new item factors).

This function allows using the same API while avoiding model refitting or deep copies of data by simply swapping the matrices, IDs, and hyperparameters as necessary.

The resulting object can be used with the same functions as the original, such as topN or factors, but any mention of "user" in the functions will now mean "items".

Usage

```
swap.users.and.items(model, precompute = TRUE)
```

Arguments

model	A collective matrix factorization model from this package - see fit_models for details.
precompute	Whether to calculate the precomputed matrices for speeding up predictions in new data.

Value

The same model object as before, but with the internal data swapped in order to make predictions about items. If passing 'precompute=TRUE', it will also generate precomputed matrices which can be used to speed up predictions.

Examples

```
library(cmfrec)

### Generate a small random matrix
n_users <- 10L
n_items <- 8L
k <- 3L
set.seed(1)
X <- matrix(rnorm(n_users*n_items), nrow=n_users)

### Factorize it
model <- CMF(X, k=k, verbose=FALSE, nthreads=1L)

### Now swap the users and items
model.swapped <- swap.users.and.items(model)

### These will now throw the same result
### (up to numerical precision)
item_factors(model, X[, 1])
factors_single(model.swapped, X[, 1])

### Swapping it again restores the original
model.restored <- swap.users.and.items(model.swapped)

### These will throw the same result
topN(model, user=2, n=3)
topN(model.restored, user=2, n=3)
```

---

topN *Calulate top-N predictions for a new or existing user*

Description

Determine top-ranked items for a user according to their predicted values, among the items to which the model was fit.

Can produce rankings for existing users (which where in the ‘X‘ data to which the model was fit) through function ‘topN‘, or for new users (which were not in the ‘X‘ data to which the model was fit, but for which there is now new data) through function ‘topN_new‘, assuming there is either ‘X‘ data, ‘U‘ data, or both (i.e. can do cold-start and warm-start rankings).

For the CMF model, depending on parameter ‘include_all_X‘, might recommend items which had only side information if their predictions are high enough.

For the ContentBased model, might be used to rank new items (not present in the ‘X‘ or ‘I‘ data to which the model was fit) given their ‘I‘ data, for new users given their ‘U‘ data. For the other models, will only rank existing items (columns of the ‘X‘ to which the model was fit) - see predict_new_items for an alternative for the other models.

Important: the model does not keep any copies of the original data, and as such, it might recommend items that were already seen/rated/consumed by the user. In order to avoid this, must manually pass the seen/rated/consumed entries to the argument ‘exclude‘ (see details below).

This method produces an exact ranking by computing all item predictions for a given user. As the number of items grows, this can become a rather slow operation - for model serving purposes, it’s usually a better idea to obtain an an approximate top-N ranking through software such as "hnsw" or "Milvus" from the calculated user factors and item factors.

Usage

```
topN(
  model,
  user = NULL,
  n = 10L,
  include = NULL,
  exclude = NULL,
  output_score = FALSE,
  nthreads = model$info$nthreads
)

topN_new(model, ...)

## S3 method for class 'CMF'
topN_new(
  model,
  X = NULL,
  X_col = NULL,
  X_val = NULL,
  U = NULL,
  U_col = NULL,
  U_val = NULL,
  U_bin = NULL,
  weight = NULL,
  n = 10L,
  include = NULL,
  exclude = NULL,
  output_score = FALSE,
```

```
  nthreads = model$info$nthreads,
  ...
)

## S3 method for class 'CMF_implicit'
topN_new(
  model,
  X = NULL,
  X_col = NULL,
  X_val = NULL,
  U = NULL,
  U_col = NULL,
  U_val = NULL,
  n = 10L,
  include = NULL,
  exclude = NULL,
  output_score = FALSE,
  nthreads = model$info$nthreads,
  ...
)

## S3 method for class 'ContentBased'
topN_new(
  model,
  U = NULL,
  U_col = NULL,
  U_val = NULL,
  I = NULL,
  n = 10L,
  include = NULL,
  exclude = NULL,
  output_score = FALSE,
  nthreads = model$info$nthreads,
  ...
)

## S3 method for class 'OMF_explicit'
topN_new(
  model,
  X = NULL,
  X_col = NULL,
  X_val = NULL,
  U = NULL,
  U_col = NULL,
  U_val = NULL,
  weight = NULL,
  exact = FALSE,
  n = 10L,
```

```
  include = NULL,
  exclude = NULL,
  output_score = FALSE,
  nthreads = model$info$nthreads,
  ...
)

## S3 method for class 'OMF_implicit'
topN_new(
  model,
  X = NULL,
  X_col = NULL,
  X_val = NULL,
  U = NULL,
  U_col = NULL,
  U_val = NULL,
  n = 10L,
  include = NULL,
  exclude = NULL,
  output_score = FALSE,
  nthreads = model$info$nthreads,
  ...
)
```

Arguments

model	A collective matrix factorization model from this package - see fit_models for details.
user	User (row of ‘X‘) for which to rank items. If ‘X‘ to which the model was fit was a ‘data.frame‘, should pass an ID matching to the first column of ‘X‘ (the user indices), otherwise should pass a row number for ‘X‘, with numeration starting at 1. This is optional for the MostPopular model, but must be passed for all others. For making recommendations about new users (that were not present in the ‘X‘ to which the model was fit), should use ‘topN_new‘ and pass either ‘X‘ or ‘U‘ data. For example usage, see the main section fit_models.
n	Number of top-predicted items to output.
include	If passing this, will only make a ranking among the item IDs provided here. See the documentation for ‘user‘ for how the IDs should be passed. This should be an integer or character vector, or alternatively, as a sparse vector from the ‘Matrix‘ package (inheriting from class ‘sparseVector‘), from which the non-missing entries will be taken as those to include. Cannot be used together with ‘exclude‘.
exclude	If passing this, will exclude from the ranking all the item IDs provided here. See the documentation for ‘user‘ for how the IDs should be passed. This should be an integer or character vector, or alternatively, as a sparse vector from the

	'Matrix' package (inheriting from class 'sparseVector'), from which the non-missing entries will be taken as those to exclude. Cannot be used together with 'include'.
output_score	Whether to also output the predicted values, in addition to the indices of the top-predicted items.
nthreads	Number of parallel threads to use.
...	Not used.
X	'X' data for a new user for which to make recommendations, either as a numeric vector (class 'numeric'), or as a sparse vector from package 'Matrix' (class 'dsparseVector'). If the 'X' to which the model was fit was a 'data.frame', the column/item indices will have been reindexed internally, and the numeration can be found under 'model$info$item_mapping'. Alternatively, can instead pass the column indices and values and let the model reindex them (see 'X_col' and 'X_val'). Should pass at most one of 'X' or 'X_col'+'X_val'. Dense 'X' data is not supported for 'CMF_implicit' or 'OMF_implicit'.
X_col	'X' data for a new user for which to make recommendations, in sparse vector format, with 'X_col' denoting the items/columns which are not missing. If the 'X' to which the model was fit was a 'data.frame', here should pass IDs matching to the second column of that 'X', which will be reindexed internally. Otherwise, should have column indices with numeration starting at 1 (passed as an integer vector). Should pass at most one of 'X' or 'X_col'+'X_val'.
X_val	'X' data for a new user for which to make recommendations, in sparse vector format, with 'X_val' denoting the associated values to each entry in 'X_col' (should be a numeric vector of the same length as 'X_col'). Should pass at most one of 'X' or 'X_col'+'X_val'.
U	'U' data for a new user for which to make recommendations, either as a numeric vector (class 'numeric'), or as a sparse vector from package 'Matrix' (class 'dsparseVector'). Alternatively, if 'U' is sparse, can instead pass the indices of the non-missing columns and their values separately (see 'U_col'). Should pass at most one of 'U' or 'U_col'+'U_val'.
U_col	'U' data for a new user for which to make recommendations, in sparse vector format, with 'U_col' denoting the attributes/columns which are not missing. Should have numeration starting at 1 (should be an integer vector). Should pass at most one of 'U' or 'U_col'+'U_val'.
U_val	'U' data for a new user for which to make recommendations, in sparse vector format, with 'U_val' denoting the associated values to each entry in 'U_col' (should be a numeric vector of the same length as 'U_col'). Should pass at most one of 'U' or 'U_col'+'U_val'.
U_bin	Binary columns of 'U' for a new user for which to make recommendations, on which a sigmoid transformation will be applied. Should be passed as a numeric vector. Note that 'U' and 'U_bin' are not mutually exclusive.
weight	(Only for the explicit-feedback models) Associated weight to each non-missing observation in 'X'. Must have the same number of entries as 'X' - that is, if passing a dense vector of length 'n', 'weight' should be a numeric vector of length 'n' too, if passing a sparse vector, should have a lenght corresponding to the number of non-missing elements.

`I` (Only for the ‘ContentBased‘ model) New ‘I‘ data to rank for the given user, with rows denoting new columns of the ‘X‘ matrix. Can be passed in the following formats:

- A sparse COO/triplets matrix, either from package ‘Matrix‘ (class ‘dgTMatrix‘), or from package ‘SparseM‘ (class ‘matrix.coo‘).
- A sparse matrix in CSR format, either from package ‘Matrix‘ (class ‘dgRMatrix‘), or from package ‘SparseM‘ (class ‘matrix.csr‘). Passing the input as CSR is faster than COO as it will be converted internally.
- A sparse row vector from package ‘Matrix‘ (class ‘dsparseVector‘).
- A dense matrix from base R (class ‘matrix‘), with missing entries set as NA.
- A dense vector from base R (class ‘numeric‘).
- A ‘data.frame‘.

When passing ‘I‘, the item indices in ‘include‘, ‘exclude‘, and in the resulting output refer to rows of ‘I‘, and the ranking will be made only among the rows of ‘I‘ (that is, they will not be compared against the old ‘X‘ data).

`exact` (In the ‘OMF_explicit‘ model) Whether to calculate ‘A‘ and ‘Am‘ with the regularization applied to ‘A‘ instead of to ‘Am‘ (if using the L-BFGS method, this is how the model was fit). This is usually a slower procedure.

Details

Be aware that this function is multi-threaded. As such, if a large batch of top-N predictions is to be calculated in parallel for different users (through e.g. ‘mclapply‘ or similar), it’s recommended to decrease the number of threads in the model to 1 (e.g. ‘model$info$nthreads <- 1L‘) and to set the number of BLAS threads to 1 (through e.g. ‘RhpcBLASctl‘ or environment variables).

For better cold-start recommendations with CMF_implicit, one can also add item biases by using the ‘CMF‘ model with parameters that would mimic ‘CMF_implicit‘ plus the biases.

Value

If passing ‘output_score=FALSE‘ (the default), will output the indices of the top-predicted elements. If passing ‘output_score=TRUE‘, will pass a list with two elements:

- ‘item‘: The indices of the top-predicted elements.
- ‘score‘: The predicted value for each corresponding element in ‘item‘.

If the ‘X‘ to which the model was fit was a ‘data.frame‘ (and unless passing ‘I‘), the item indices will be taken from the same IDs in ‘X‘ (its second column) - but be aware that in this case they will usually be returned as ‘character‘. Otherwise, will return the indices of the top-predicted columns of ‘X‘ (or rows of ‘I‘ if passing it) with numeration starting at 1.

See Also

factors_single predict.cmfrec predict_new

Index