cmfrec Documentation

David Cortes

Aug 10, 2018

Contents:

This is the documentation page for the python package *cmfrec*. For more details, see the project's GitHub page:
https://www.github.com/david-cortes/cmfrec/

CHAPTER 1

Installation

Package is available on PyPI, can be installed with

```
pip install cmfrec
```

class cmfrec.**CMF**(*k=30*, *k_main=0*, *k_user=0*, *k_item=0*, *w_main=1.0*, *w_user=1.0*, *w_item=1.0*, *reg_param=0.0001*, *offsets_model=False*, *nonnegative=False*, *maxiter=1000*, *standardize_err=True*, *reweight=False*, *reindex=True*, *center_ratings=True*, *add_user_bias=True*, *add_item_bias=True*, *center_user_info=False*, *center_item_info=False*, *user_info_nonneg=False*, *item_info_nonneg=False*, *keep_data=True*, *save_folder=None*, *produce_dicts=True*, *random_seed=None*, *verbose=True*)

Bases: object

Collective matrix factorization model for recommenders systems with explicit data and side info.

Fits a collective matrix factorization model to ratings data along with item and/or user side information, by factorizing all matrices with common (shared) factors, e.g.: X~=AB' and I~=BC'

By default, the function to minimize is as follows: L = w_main*norm(X- AB' - Ubias_{u} - Ibias_{i})^2/|X| + w_item*norm(I-BC')^2/|I| + w_user*norm(U-AD')^2/|U|

with added regularization as follows: L += reg_param*(norm(A)^2+norm(B)^2+norm(C)^2+norm(D)^2)

And user/item biases

Where:

X is the ratings matrix (considering only non-missing entries). I is the item-attribute matrix (only supports dense inputs). U is the user-attribute matrix (only supports dense inputs). A, B, C, D are lower-dimensional matrices (the model parameters). |X|, |I|, |U| are the number non-missing of entries in each matrix. The matrix products might not use all the rows/columns of these matrices at each factorization (this is controlled with k_main, k_item and k_user). Ubias_{u} and Ibias_{i} are user and item biases, defined for each user and item ID.

** Be aware that, due to the regularization part, this formula as it is implies that larger datasets require lower regularization. You can use 'standardize_err=False' and set w1/w2/w3 to avoid this.**

Alternatively, it can also fit an additive model with "offsets", similar in spirit to [2] (see references):

L = norm(X - (A+UC)(B+ID)' - Ubias_{u} - Ibias_{i})^2 + reg_param*(norm(A)^2+norm(B)^2+norm(C)^2+norm(D)^2)

This second model requires more iterations to fit (takes more time), and doesn't support missing value imputation, but it oftentimes provides better results, especially for cold-start recommendations and if the side information (users and items) is mostly binary columns.

In both cases, the model is fit with full L-BFGS updates (not stochastic gradient descent or stochastic Newton), i.e. it calculates errors for the whole data and updates all model parameters at once during each iteration. This usually leads to better local optima and less need for parameter tuning, but at the cost of less scalability. The L-BFGS implementation used is from SciPy, which is not entirely multi-threaded, so if your CPU has many cores or you have many GPUs, you can expect an speed up from parallelization on the calculations of the objective function and the gradient, but not so much on the calculations that happen inside L-BFGS.

By default, the number of iterations is set at 1000, but for smaller datasets and for the offsets model, this might not reach convergence when using high regularization.

If passing reindex=True, it will internally reindex all user and item IDs. Your data will not require reindexing if the IDs for users and items in the input data frame passed to .fit meet the following criteria:

1. Are all integers.
2. Start at zero.
3. Don't have any enumeration gaps, i.e. if there is a user '4', user '3' must also be there.

Adding side information about entries for which there are no ratings or vice-versa can still help to improve factorizations, but the more elements they have in common, the better.

If passing reindex=False, then the matrices with side information (user and item attributes) must have exactly the same number of rows as the number of users/items present in the ratings data, in which case there cannot be entries (users or items) missing from the ratings and present in the side information or vice-versa.

For missing entries in the user and item attributes, use numpy.nan. These will not be taken into account in the optimization procedure. If there is no information for use user or item, leave that row out altogether instead of filling with NAs. In the offsets model, missing entries will be automatically filled with zeros, so it's recommended to perform imputation beforehand for it.

Can also produce non-negative matrix factorization (including the user and item attributes), but if using user and/or item biases, these will not be constrained to be non-negative.

For the regular model, if there are binary columns in the data, it can apply a sigmoid transformation to the approximations from the factorization, but these will still be taken as a squared loss with respect to the original 0/1 values, so as to make the loss comparable to that of the other columns. Be sure to pass the names of the binary columns to '.fit', or the indexes or the columns when using 'reindex=False'.

Note: Be aware that the data passed to '.fit' will be modified inplace (e.g. reindexed). Make a copy of your data beforehand if you require it. If you plan to do any hyper-parameter tuning through cross-validation, you should reindex your data beforehand and call the CMF constructur with 'index=False'.

Note: The API contains parameters for both an item-attribute matrix and a user-attribute matrix, but you can fit the model to data with only one or none of them. Parameters corresponding to the factorization of a matrix for which no data is passed will be ignored.

Note: The model allows to make recommendations for users and items for which there is data about their attributes but no ratings. The quality of these predictions might however not be good, especially if you set

k_main > 0. It is highly recommended to center your ratings if you plan on making predictions for user/items that were not in the training set.

Parameters

- **k** (*int*) – Number of common (shared) latent factors to use.
- **k_main** (*int*) – Number of additional (non-shared) latent factors to use for the ratings factorization. Ignored for the offsets model.
- **k_user** (*int*) – Number of additional (non-shared) latent factors to use for the user attributes factorization. Ignored for the offsets model.
- **k_item** (*int*) – Number of additional (non-shared) latent factors to use for the item attributes factorization. Ignored for the offsets model.
- **w_main** (*float*) – Weight to assign to the (mean) root squared error in factorization of the ratings matrix. Ignored for the offsets model.
- **w_user** (*float*) – Weight to assign to the (mean) root squared error in factorization of the user attributes matrix. Ignored for the offsets model.
- **w_item** (*float*) – Weight to assign to the (mean) root squared error in factorization of the item attributes matrix. Ignored for the offsets model.
- **reg_param** (*float or tuple of floats*) – Regularization parameter for each matrix, in this order: 1) User-Factor, 2) Item-Factor, 3) User bias, 4) Item-bias, 5) UserAttribute-Factor, 6) ItemAttribute-Factor.
- **offsets_model** (*bool*) – Whether to fit the alternative model formulation with offsets (see description).
- **nonnegative** (*bool*) – Whether the resulting low-rank matrices (A and B) should have all non-negative entries. Forced to False when passing 'center_ratings=True'.
- **maxiter** (*int*) – Maximum number of iterations for which to run the optimization procedure. Recommended to use a higher number for the offsets model.
- **standardize_err** (*bool*) – Whether to divide the sum of squared errors from each factorized matrix by the number of non-missing entries. Setting this to False requires far larger regularization parameters. Note that most research papers don't standardize the errors, so if you try to reproduce some paper with specific parameters, you might want to set this to True. Forced to False when passing 'reweight=True'.
- **reweight** (*bool*) – Whether to automatically reweight the errors of each matrix factorization so that they get similar influence, accounting for the number of elements in each and the magnitudes of the entries (appplies in addition to weights passed as w_main, w_item and w_user). This is done by calculating the initial sum of squared errors with randomly initialized factor matrices, but it's not guaranteed to be a good criterion. It might be better to scale the entries of either the ratings or the attributes matrix so that they are in a similar scale (e.g. if the ratings are in [1,5], the attributes should ideally be in the same range and not [-10^3,10^3]). Ignored for the offsets model.
- **reindex** (*bool*) – Whether to reindex data internally (assign own IDs) - see description above.
- **center_ratings** (*bool*) – Whether to substract the mean rating from the ratings before fitting the model. Will be force to True if passing 'add_user_bias=True' or 'add_item_bias=True'.
- **user_bias** (*bool*) – Whether to use user biases (one per user) as additional model parameters.

- **item_bias** (*bool*) – Whether to use item biases (one per item) as additional model parameters.
- **center_user_info** (*bool*) – Whether to center the user attributes by substracting the mean from each column.
- **center_item_info** (*bool*) – Whether to center the item attributes by substracting the mean from each column.
- **user_info_nonneg** (*bool*) – Whether the user_attribute-factor matrix (C) should have all non-negative entries. Forced to false when passing 'center_user_info=True'.
- **item_info_nonneg** (*bool*) – Whether the item_attribute-factor matrix (D) should have all non-negative entries. Forced to false when passing 'center_item_info=True'
- **keep_data** (*bool*) – Whether to keep information about which user was associated with each item in the training set, so as to exclude those items later when making Top-N recommendations.
- **save_folder** (*str or None*) – Folder where to save all model parameters as csv files.
- **produce_dicts** (*bool*) – Whether to produce Python dictionaries for users and items, which are used by the prediction API of this package. You can still predict without them, but it might take some additional miliseconds (or more depending on the number of users and items).
- **random_seed** (*int or None*) – Random seed to use when starting the parameters.
- **verbose** (*bool*) – Whether to display convergence messages frol L-BFGS. If running it from an IPython notebook, these will be printed in the console in which the notebook is running, but not on the output of the cell within the notebook.

Variables

- **A** (*array (nitems, k_main + k + k_user)*) – Matrix with the user-factor attributes, containing columns from both factorizations. If you wish to extract only the factors used for predictons, slice it like this: A[:,:k_main+k]
- **B** (*array (nusers, k_main + k + k_item)*) – Matrix with the item-factor attributes, containing columns from both factorizations. If you wish to extract only the factors used for predictons, slice it like this: B[:,:k_main+k]
- **C** (*array (k + k_item, item_dim)*) – Matrix of factor-item_attribute. Will have the columns that correspond to binary features in a separate attribute.
- **D** (*array (k_user + k, user_dim)*) – Matrix of factor-user_attribute. Will have the columns that correspond to binary features in a separate attribute.
- **C_bin** (*array (k + k_item, item_dim_bin)*:) – Part of the C matrix that corresponds to binary columns in the item data. Non-negativity constraints will not apply to this matrix.
- **D_bin** (*array (k_user + k, user_dim_bin)*) – Part of the D matrix that corresponds to binary columns in the user data. Non-negativity constraints will not apply to this matrix.
- **add_user_bias** (*array (nusers,)*) – User biases determined by the model
- **add_item_bias** (*array (nitems,)*) – Item biases determined by the model
- **user_mapping** (*array (nusers,)*) – ID of the user (as passed to .fit) represented by each row of A.

- **item_mapping** (*array (nitems,)*) – ID of the item (as passed to .fit) represented by each row of B.
- **user_dict** (*dict (nusers)*) – Dictionary with the mapping between user IDs (as passed to .fit) and rows of A.
- **item_dict** (*dict (nitems)*) – Dictionary with the mapping between item IDs (as passed to .fit) and rows of B.
- **is_fitted** (*bool*) – Whether the model has been fit to some data.
- **global_mean** (*float*) – Global mean of the ratings.
- **user_arr_means** (*array (user_dim,)*) – Column means of the user side information matrix.
- **item_arr_means** (*array (item_dim,)*) – Column means of the item side information matrix.

References

[1] Relational learning via collective matrix factorization (A. Singh, 2008) [2] Collaborative topic modeling for recommending scientific articles (C. Wang, D. Blei, 2011)

add_item(*new_id*, *attributes*, *reg='auto'*)

Adds a new item vector according to its attributes, in order to make predictions for it

In the regular collective factorization model without non-negativity constraints and without binary columns, will calculate the latent factors vector by its closed-form solution, which is fast. In the offsets model, the latent factors vector is obtained by a simple matrix product, so it will be even faster. However, if there are non-negativity constraints and/or binary columns, there is no closed form solution, and it will be calculated via gradient-based optimization, so it will take longer and shouldn't be expected to work in 'real time'.

Note: For better quality cold-start recommendations, center your ratings data, use high regularization, assign large weights to the factorization of side information, and don't use large values for number of latent factors that are specific for some factorization.

Note: If you pass and ID that is of a different type (str, int, obj, etc.) than the IDs of the data that was passed to .fit, the internal indexes here might break and some of the prediction functionality might stop working. Be sure to pass IDs of the same type. The type of the ID will be forcibly converted to try to avoid this, but you might still run into problems.

Parameters

- **new_id** (*obj*) – ID of the new item. Ignored when called with 'reindex=False', in which case it will assign it ID = nitems_train + 1.
- **attributes** (*array (item_dim,)*) – Attributes of this item (side information)
- **reg** (*float or str 'auto'*) – Regularization parameter for these new attributes. If set to 'auto', will use the same regularization parameter that was set for the item-factor matrix.

Returns **Success** – Returns true if the operation completes successfully

Return type bool

add_user (*new_id*, *attributes*, *reg='auto'*)

Adds a new user vector according to its attributes, in order to make predictions for her

In the regular collective factorization model without non-negativity constraints and without binary columns, will calculate the latent factors vector by its closed-form solution, which is fast. In the offsets model, the latent factors vector is obtained by a simple matrix product, so it will be even faster. However, if there are non-negativity constraints and/or binary columns, there is no closed form solution, and it will be calculated via gradient-based optimization, so it will take longer and shouldn't be expected to work in 'real time'.

Note: For better quality cold-start recommendations, center your ratings data, use high regularization, assign large weights to the factorization of side information, and don't use large values for number of latent factors that are specific for some factorization.

Note: If you pass and ID that is of a different type (str, int, obj, etc.) than the IDs of the data that was passed to .fit, the internal indexes here might break and some of the prediction functionality might stop working. Be sure to pass IDs of the same type. The type of the ID will be forcibly converted to try to avoid this, but you might still run into problems.

Parameters

- **new_id** (*obj*) – ID of the new user. Ignored when called with 'reindex=False', in which case it will assign it ID = nusers_train + 1.
- **attributes** (*array (user_dim,)*) – Attributes of this user (side information)
- **reg** (*float or str 'auto'*) – Regularization parameter for these new attributes. If set to 'auto', will use the same regularization parameter that was set for the user-factor matrix.

Returns **Success** – Returns true if the operation completes successfully

Return type bool

fit (*ratings*, *user_info=None*, *item_info=None*, *cols_bin_user=None*, *cols_bin_item=None*)

Fit the model to ratings data and item/user side info, using L-BFGS

Note: Be aware that the data passed to '.fit' will be modified inplace (e.g. reindexed). Make a copy of your data beforehand if you require it (e.g. using the "deepcopy" function from the "copy" module).

Parameters

- **ratings** (*pandas data frame or array (nobs, 3)*) – Ratings data to which to fit the model. If a pandas data frame, must contain the columns 'UserId','ItemId' and 'Rating'. Optionally, it might also contain a column 'Weight'. If a numpy array, will take the first 4 columns in that order If a list of tuples, must be in the format (UserId,ItemId,Rating,[Weight]) (will be coerced to data frame)
- **user_info** (*pandas data frame or numpy array (nusers, nfeatures_user)*) – Side information about the users (i.e. their attributes, as a table). Must contain a column called 'UserId'. If called with 'reindex=False', must be a numpy array, with rows corresponding to user ID numbers and columns to user attributes.

- **item_info** (*pandas data frame or numpy array (nitems, nfeatures_item)*) – Side information about the items (i.e. their attributes, as a table). Must contain a column named ItemId. If called with 'reindex=False', must be a numpy array, with rows corresponding to item ID numbers and columns to item attributes.
- **cols_bin_user** (*array or list*) – Columns of user_info that are binary (only take values 0 or 1). Will apply a sigmoid function to the factorized approximations of these columns. Ignored when called with 'offsets_model=True'.
- **cols_bin_item** (*array or list*) – Columns of item_info that are binary (only take values 0 or 1). Will apply a sigmoid function to the factorized approximations of these columns. Ignored when called with 'offsets_model=True'.

Returns **self** – Copy of this object

Return type obj

predict(*user*, *item*)
Predict ratings for combinations of users and items

Note: You can either pass an individual user and item, or arrays representing tuples (UserId, ItemId) with the combinatinons of users and items for which to predict (one row per prediction).

Note: If you pass any user/item which was not in the training set, the prediction for it will be NaN.

Parameters

- **user** (*array-like (npred,) or obj*) – User(s) for which to predict each item.
- **item** (*array-like (npred,) or obj*) – Item(s) for which to predict for each user.

topN(*user*, *n=10*, *exclude_seen=True*, *items_pool=None*)
Recommend Top-N items for a user

Outputs the Top-N items according to score predicted by the model. Can exclude the items for the user that were associated to her in the training set, and can also recommend from only a subset of user-provided items.

Parameters

- **user** (*obj*) – User for which to recommend.
- **n** (*int*) – Number of top items to recommend.
- **exclude_seen** (*bool*) – Whether to exclude items that were associated to the user in the training set.
- **items_pool** (*None or array*) – Items to consider for recommending to the user.

Returns **rec** – Top-N recommended items.

Return type array (n,)

topN_cold(*attributes*, *n=10*, *reg='auto'*, *items_pool=None*)
Recommend Top-N items for a user that was not in the training set.

In the regular collective factorization model without non-negativity constraints and without binary columns, will calculate the latent factors vector by its closed-form solution, which is fast. In the offsets model, the latent factors vector is obtained by a simple matrix product, so it will be even faster. However,

if there are non-negativity constraints and/or binary columns, there is no closed form solution, and it will be calculated via gradient-based optimization, so it will take longer and shouldn't be expected to work in 'real time'.

Note: For better quality cold-start recommendations, center your ratings data, use high regularization, assign large weights to the factorization of side information, and don't use large values for number of latent factors that are specific for some factorization.

Parameters

- **attributes** (*array (user_dim,)*) – Attributes of the user. Columns must be in the same order as was passed to '.fit', but without the ID column.
- **n** (*int*) – Number of top items to recommend.
- **reg** (*float or str 'auto'*) – Regularization parameter for these new attributes. If set to 'auto', will use the same regularization parameter that was set for the user-factor matrix.
- **items_pool** (*None or array*) – Items to consider for recommending to the user.

Returns **rec** – Top-N recommended items.

Return type array (n,)

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Index