Package 'recometrics'

February 20, 2023

Type Package

Title Evaluation Metrics for Implicit-Feedback Recommender Systems

Version 0.1.6-3

Author David Cortes

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

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

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

Description Calculates evaluation metrics for implicit-feedback recommender systems that are based on low-rank matrix factorization models, given the fitted model matrices and data, thus allowing to compare models from a variety of libraries.
Metrics include P@K (precision-at-k, for top-K recommendations), R@K (recall at k), AP@K (average precision at k), NDCG@K (normalized discounted cumulative gain at k), Hit@K (from which the 'Hit Rate' is calculated), RR@K (reciprocal rank at k, from which the 'MRR' or 'mean reciprocal rank' is calculated), ROC-AUC (area under the receiver-operating characteristic curve), and PR-AUC (area under the precision-recall curve).
These are calculated on a per-user basis according to the ranking of items induced by the model, using efficient multi-threaded routines. Also provides functions for creating train-test splits for model fitting and evaluation.

LinkingTo Rcpp, float

Imports Rcpp (>= 1.0.1), Matrix (>= 1.3-4), MatrixExtra (>= 0.1.6), float, RhpcBLASctl, methods

Suggests recommenderlab (>= 0.2-7), cmfrec (>= 3.2.0), data.table, knitr, rmarkdown, kableExtra, testthat

VignetteBuilder knitr

License BSD_2_clause + file LICENSE

RoxygenNote 7.1.1

StagedInstall TRUE

Biarch TRUE

NeedsCompilation yes

Repository CRAN

Date/Publication 2023-02-19 23:00:02 UTC

R topics documented:

`calc.reco.metrics` *Calculate Recommendation Quality Metrics*

Description

Calculates recommendation quality metrics for implicit-feedback recommender systems (fit to user-item interactions data such as "number of times that a user played each song in a music service") that are based on low-rank matrix factorization or for which predicted scores can be reduced to a dot product between user and item factors/components.

These metrics are calculated on a per-user basis, by producing a ranking of the items according to model predictions (in descending order), ignoring the items that are in the training data for each user. The items that were not consumed by the user (not present in ‘X_train‘ and not present in ‘X_test‘) are considered "negative" entries, while the items in ‘X_test‘ are considered "positive" entries, and the items present in ‘X_train‘ are ignored for these calculations.

The metrics that can be calculated by this function are:

- ‘P@K‘ ("precision-at-k"): denotes the proportion of items among the top-K recommended (after excluding those that were already in the training data) that can be found in the test set for that user:
  $P@K = \frac{1}{k} \sum_{i=1}^{k} r_i \in \mathcal{T}$
  This is perhaps the most intuitive and straightforward metric, but it can present a lot of variation between users and does not take into account aspects such as the number of available test items or the specific ranks at which they are shown.
- ‘TP@K‘ (truncated precision-at-k): a truncated or standardized version of the precision metric, which will divide instead by the minimum between ‘k‘ and the number of test items:
  $TP@K = \frac{1}{\min\{k, \mathcal{T}\}} \sum_{i=1}^{k} r_i \in \mathcal{T}$
  Note: many papers and libraries call this the "P@K" instead. The "truncated" prefix is a non-standard nomenclature introduced here to differentiate it from the P@K metric that is calculated by this and other libraries.
- ‘R@K‘ ("recall-at-k"): proportion of the test items that are retrieved in the top-K recommended list. Calculation is the same as precision, but the division is by the number of test items instead of ‘k‘:
  $R@K = \frac{1}{|\mathcal{T}|} \sum_{i=1}^{k} r_i \in \mathcal{T}$
- ‘AP@K‘ ("average precision-at-k"): precision and recall look at all the items in the top-K equally, whereas one might want to take into account also the ranking within this top-K list, for which this metric comes in handy. "Average Precision" tries to reflect the precisions that would be obtained at different recalls:
  $AP@K = \frac{1}{|\mathcal{T}|} \sum_{i=1}^{k} (r_i \in \mathcal{T}) \cdot P@i$

This is a metric which to some degree considers precision, recall, and rank within top-K. Intuitively, it tries to approximate the are under a precision-recall tradeoff curve.

The average of this metric across users is known as "Mean Average Precision" or "MAP@K".

IMPORTANT: many authors define AP@K differently, such as dividing by the minimum between ‘k‘ and the number of test items instead, or as the average for P@1..P@K (either as-is or stopping after already retrieving all the test items) - here, the second version is offered as different metric instead. This metric is likely to be a source of mismatches when comparing against other libraries due to all the different defintions used by different authors.

- ‘TAP@K‘ (truncated average precision-at-k): a truncated version of the AP@K metric, which will instead divide it by the minimum between ‘k‘ and the number of test items.

  Many other papers and libraries call this the "average precision" instead.

- ‘NDCG@K‘ (normalized discounted cumulative gain at K): a ranking metric calculated by first determining the following:

  $\sum_{i=1}^{k} \frac{C_i}{log_2(i+1)}$

  Where C_i denotes the confidence score for an item (taken as the value in ‘X_test‘ for that item), with ‘i‘ being the item ranked at a given position for a given user according to the model. This metric is then standardized by dividing by the maximum achievable such score given the test data.

  Unlike the other metrics:

  - It looks not only at the presence or absence of items, but also at their confidence score.
  - It can handle data which contains "dislikes" in the form of negative values (see caveats below).

  If there are only non-negative values in ‘X_test‘, this metric will be bounded between zero and one.

  A note about negative values: the NDCG metric assumes that all the values are non-negative. This implementation however can accommodate situations in which a very small fraction of the items have negative values, in which case: (a) it will standardize the metric by dividing by a number which does not consider the negative values in its sum; (b) it will be set to ‘NA‘ if there are no positive values. Be aware however that NDCG loses some of its desirable properties in the presence of negative values.

- ‘Hit@K‘ ("hit-at-k"): indicates whether any of the top-K recommended items can be found in the test set for that user. The average across users is typically referred to as the "Hit Rate".

  This is a binary metric (it is either zero or one, but can also be ‘NA‘ when it is not possible to calculate, just like the other metrics).

- ‘RR@K‘ ("reciprocal-rank-at-k"): inverse rank (one divided by the rank) of the first item among the top-K recommended that is in the test data. The average across users is typically referred to as the "Mean Reciprocal Rank" or MRR.

  If none of the top-K recommended items are in the test data, will be set to zero.

- ‘ROC-AUC‘ (area under the receiver-operating characteristic curve): see the Wikipedia entry for details. This metric considers the full ranking of items rather than just the top-K. It is bounded between zero and one, with a value of 0.5 corresponding to a random order and a value of 1 corresponding to a perfect ordering (i.e. every single positive item has a higher predicted score than every single negative item).

  Be aware that there are different ways of calculating AUC, with some methods having higher precision than others. This implementation uses a fast formula which implies dividing two large numbers, and as such might not be as precise to the same number of decimals as the trapezoidal method used by e.g. scikit-learn.

- ‘PR-AUC‘ (area under the precision-recall curve): while ROC AUC provides an overview of the overall ranking, one is typically only interested in how well it retrieves test items within top ranks, and for this the area under the precision-recall curve can do a better job at judging rankings, albeit the metric itself is not standardized, and under the worst possible ranking, it does not evaluate to zero.

  The metric is calculated using the fast but not-so-precise rectangular method, whose formula corresponds to the AP@K metric with K=N. Some papers and libraries call this the average of this metric the "MAP" or "Mean Average Precision" instead (without the "@K").

Metrics can be calculated for a given value of ‘k‘ (e.g. "P@3"), or for values ranging from 1 to ‘k‘ (e.g. ["P@1", "P@2", "P@3"]).

This package does **NOT** cover other more specialized metrics. One might also want to look at other indicators such as:

- Metrics that look at the rareness of the items recommended (to evaluate so-called "serendipity").
- Metrics that look at "discoverability".
- Metrics that take into account the diversity of the ranked lists.

Usage

```
calc.reco.metrics(
  X_train,
  X_test,
  A,
  B,
  k = 5L,
  item_biases = NULL,
  as_df = TRUE,
  by_rows = FALSE,
  sort_indices = TRUE,
  precision = TRUE,
  trunc_precision = FALSE,
  recall = FALSE,
  average_precision = TRUE,
  trunc_average_precision = FALSE,
  ndcg = TRUE,
  hit = FALSE,
  rr = FALSE,
  roc_auc = FALSE,
  pr_auc = FALSE,
  all_metrics = FALSE,
  rename_k = TRUE,
  break_ties_with_noise = TRUE,
  min_pos_test = 1L,
  min_items_pool = 2L,
  consider_cold_start = TRUE,
  cumulative = FALSE,
  nthreads = parallel::detectCores(),
  seed = 1L
)
```

Arguments

`X_train` Training data for user-item interactions, with users denoting rows, items denoting columns, and values corresponding to confidence scores. Entries in ‘X_train‘ and ‘X_test‘ for each user should not intersect (that is, if an item is the training data as a non-missing entry, it should not be in the test data as non-missing, and vice versa).

Should be passed as a sparse matrix in CSR format (class ‘dgRMatrix‘ from package ‘Matrix‘, can be converted to that format using ‘MatrixExtra::as.csr.matrix‘). Items not consumed by the user should not be present in this matrix.

Alternatively, if there is no training data, can pass ‘NULL‘, in which case it will look only at the test data.

This matrix and ‘X_test‘ are not meant to contain negative values, and if ‘X_test‘ does contain any, it will still be assumed for all metrics other than NDCG that such items are deemed better for the user than the missing/zero-valued items (that is, implicit feedback is not meant to signal dislikes).

`X_test` Test data for user-item interactions. Same format as ‘X_train‘.

`A` The user factors. If the number of users is ’m’ and the number of factors is ’p’, should have dimension ‘[p, m]‘ if passing ‘by_rows=FALSE‘ (the default), or dimension ‘[m, p]‘ if passing ‘by_rows=TRUE‘ (in wich case it will be internally transposed due to R’s column-major storage order). Can be passed as a dense matrix from base R (class ‘matrix‘), or as a matrix from package float (class ‘float32‘) - if passed as ‘float32‘, will do the calculations in single precision (which is faster and uses less memory) and output the calculated metrics as ‘float32‘ arrays.

It is assumed that the model score for a given item ‘j‘ for user ‘i‘ is calculated as the inner product or dot product between the corresponding vectors $\mathbf{a}_i \cdot \mathbf{b}_j$ (columns ‘i‘ and ‘j‘ of ‘A‘ and ‘B‘, respectively, when passing ‘by_rows=FALSE‘), with higher scores meaning that the item is deemed better for that user, and the top-K recommendations produced by ranking these scores in descending order.

Alternatively, for evaluation of non-personalized models, can pass ‘NULL‘ here and for ‘B‘, in which case ‘item_biases‘ must be passed.

`B` The item factors, in the same format as ‘A‘.

`k` The number of top recommendations to consider for the metrics (as in "precision-at-k" or "P@K").

`item_biases` Optional item biases/intercepts (fixed base score that is added to the predictions of each item). If present, it will append them to ‘B‘ as an extra factor while adding a factor of all-ones to ‘A‘.

Alternatively, for non-personalized models which have only item-by-item scores, can pass ‘NULL‘ for ‘A‘ and ‘B‘ while passing only ‘item_biases‘.

`as_df` Whether to output the result as a ‘data.frame‘. If passing ‘FALSE‘, the results will be returned as a list of vectors or matrices depending on what is passed for ‘cumulative‘. If ‘A‘ and ‘B‘ are passed as ‘float32‘ matrices, the resulting ‘float32‘ arrays will be converted to base R’s arrays in order to be able to create a ‘data.frame‘.

`by_rows` Whether the latent factors/components are ordered by rows, in which case they will be transposed beforehand (see documentation for ‘A‘).

`sort_indices` Whether to sort the indices of the ‘X‘ data in case they are not sorted already. Skipping this step will make it faster and will make it consume less memory.

If the ‘X_train‘ and ‘X_test‘ matrices were created using functions from the ‘Matrix‘ package such as ‘Matrix::spMatrix‘ or ‘Matrix::Matrix‘, the indices will always be sorted, but if creating it manually through S4 methods or as the output of other software, the indices can end up unsorted.

`precision` Whether to calculate precision metrics or not.

`trunc_precision`

Whether to calculate truncated precision metrics or not. Note that this is output as a separate metric from "precision" and they are not mutually exclusive options.

`recall` Whether to calculate recall metrics or not.

`average_precision`

Whether to calculate average precision metrics or not.

`trunc_average_precision`

Whether to calculate truncated average precision metrics or not. Note that this is output as a separate metric from "average_precision" and they are not mutually exclusive options.

`ndcg` Whether to calculate NDCG (normalized discounted cumulative gain) metrics or not.

`hit` Whether to calculate Hit metrics or not.

`rr` Whether to calculate RR (reciprocal rank) metrics or not.

`roc_auc` Whether to calculate ROC-AUC (area under the ROC curve) metrics or not.

`pr_auc` Whether to calculate PR-AUC (area under the PR curve) metrics or not.

`all_metrics` Passing ‘TRUE‘ here is equivalent to passing ‘TRUE‘ to all the calculable metrics.

`rename_k` If passing ‘as_df=TRUE‘ and ‘cumulative=FALSE‘, whether to rename the ’k’ in the resulting column names to the actual value of ’k’ that was used (e.g. "p_at_k" -> "p_at_5").

`break_ties_with_noise`

Whether to add a small amount of noise ‘~Uniform(-10^-12, 10^-12)‘ in order to break ties at random, in case there are any ties in the ranking. This is not recommended unless one expects ties (can happen if e.g. some factors are set to all-zeros for some items), as it has the potential to slightly alter the ranking.

`min_pos_test` Minimum number of positive entries (non-zero entries in the test set) that users need to have in order to calculate metrics for that user. If a given user does not meet the threshold, the metrics will be set to ‘NA‘.

`min_items_pool` Minimum number of items (sum of positive and negative items) that a user must have in order to calculate metrics for that user. If a given user does not meet the threshold, the metrics will be set to ‘NA‘.

`consider_cold_start`

Whether to calculate metrics in situations in which some user has test data but no positive (non-zero) entries in the training data. If passing ‘FALSE‘ and such cases are encountered, the metrics will be set to ‘NA‘.

Will be automatically set to ‘TRUE‘ when passing ‘NULL‘ for ‘X_train‘.

`cumulative` Whether to calculate the metrics cumulatively (e.g. [P@1, P@2, P@3] if passing ‘k=3‘) for all values up to ‘k‘, or only for a single desired ‘k‘ (e.g. only P@3 if passing ‘k=3‘).

`nthreads` Number of parallel threads to use. Parallelization is done at the user level, so passing more threads than there are users will not result in a speed up. Be aware that, the more threads that are used, the higher the memory consumption.

`seed` Seed used for random number generation. Only used when passing ‘break_ties_with_noise=TRUE‘.

Details

Metrics for a given user will be set to ‘NA‘ in the following situations:

- All the rankeable items have the exact same predicted score.
- One or more of the predicted scores evaluates to ‘NA‘/‘NaN‘.
- There are only negative entries (no non-zero entries in the test data).
- The number of available items to rank (between positive and negative) is smaller than the requested ‘k‘, and the metric is not affected by the exact order within the top-K items (i.e. precision, recall, hit, will be ‘NA‘ if there's ‘k‘ or fewer items after discarding those from the training data).
- There are inconsistencies in the data (e.g. number of entries being greater than the number of columns in ‘X‘, meaning the matrices do not constitute valid CSR).
- A user does not meet the minimum criteria set by the configurable parameters for this function.
- There are only positive entries (i.e. the user already consumed all the items). In this case, "NDCG@K" will still be calculated, while the rest will be set to ‘NA‘.

The NDCG@K metric with ‘cumulative=TRUE‘ will have lower decimal precision than with ‘cumulative=FALSE‘ when using ‘float32‘ inputs - this is extremely unlikely to be noticeable in typical datasets and small ‘k‘, but for large ‘k‘ and large (absolute) values in ‘X_test‘, it might make a difference after a couple of decimal points.

Internally, it relies on BLAS function calls, so it's recommended to use R with an optimized BLAS library such as OpenBLAS or MKL for better speed - see this link for instructions on getting OpenBLAS in R for Windows (Alternatively, Microsoft's R distribution comes with MKL preinstalled).

Doing computations in float32 precision depends on the package float, and as such comes with some caveats:

- On Windows, if installing ‘float‘ from CRAN, it will use very unoptimized routines which will likely result in a slowdown compared to using regular double (numeric) type. Getting it to use an optimized BLAS library is not as simple as substituting the Rblas DLL - see the package's README for details.

- On macOS, it will use static linking for ‘float‘, thus if changing the BLAS library used by R, it will not change the float32 functions, and getting good performance out of it might require compiling it from source with ‘-march=native‘ flag.

Value

Will return the calculated metrics on a per-user basis (each user corresponding to a row):

- If passing ‘as_df=TRUE‘ (the default), the result will be a ‘data.frame‘, with the columns named according to the metric they represent (e.g. "p_at_3", see below for the other names that they can take). Depending on the value passed for ‘rename_k‘, the column names might end in "k" or in the number that was passed for ‘k‘ (e.g "p_at_3" or "p_at_k").

  If passing ‘cumulative=TRUE‘, they will have names ranging from 1 to ‘k‘.
- If passing ‘as_df=FALSE‘, the result will be a list with entries named according to each metric, with ‘k‘ as letter rather than number (‘p_at_k‘, ‘tp_at_k‘, ‘r_at_k‘, ‘ap_at_k‘, ‘tap_at_k‘, ‘ndcg_at_k‘, ‘hit_at_k‘, ‘rr_at_k‘, ‘roc_auc‘, ‘pr_auc‘), plus an additional entry with the actual ‘k‘.

  The values under each entry will be vectors if passing ‘cumulative=FALSE‘, or matrices (users corresponding to rows) if passing ‘cumulative=TRUE‘.

The ‘ROC-AUC‘ and ‘PR-AUC‘ metrics will be named just "roc_auc" and "pr_auc", since they are calculated for the full ranked predictions without stopping at ‘k‘.

Examples

```
### (See the package vignette for a better example)
library(recometrics)
library(Matrix)
library(MatrixExtra)

### Generating random data
n_users <- 10L
n_items <- 20L
n_factors <- 3L
k <- 4L
set.seed(1)
UserFactors <- matrix(rnorm(n_users * n_factors), nrow=n_factors)
ItemFactors <- matrix(rnorm(n_items * n_factors), nrow=n_factors)
X <- Matrix::rsparsematrix(n_users, n_items, .5, repr="R")
X <- abs(X)

### Generating a random train-test split
data_split <- create.reco.train.test(X, split_type="all")
X_train <- data_split$X_train
X_test <- data_split$X_test

### Calculating these metrics
### (should be poor quality, since data is random)
metrics <- calc.reco.metrics(
    X_train, X_test,
    UserFactors, ItemFactors,
    k=k, as_df=TRUE,
    nthreads=1L
)
print(metrics)
```

`create.reco.train.test` *Create Train-Test Splits of Implicit-Feedback Data*

Description

Creates train-test splits of implicit-feedback data (recorded user-item interactions) for fitting and evaluating models for recommender systems.

These splits choose "test users" and "items for a given user" separately, offering three modes of splitting the data:

- Creating training and testing sets for each user in the data (when passing ‘split_type=’all’‘).

  This is meant for cases in which the number of users is small or the users to test have already been selected (e.g. one typically does not want to create a train-test split which would leave one item for the user in the training data and zero in the test set, or would want to have other minimum criteria for the test set to be usable). Typically, one would want to take only a sub-sample of users for evaluation purposes, as calculating recommendation quality metrics can take a very long time.
- Selecting a sub-set of users for testing, for which training and testing data splits will be generated, while leaving the remainder of users with all the data for model fitting (when passing ‘split_type=’separated’‘).

  This is meant to be used for fitting a model to the remainder of the data, then generating latent factors (assuming a low-rank matrix factorization model) or top-K recommendations for the test users given their training data, and evaluating these recommendations on the test data for each user (which can be done through the function calc.reco.metrics).
- Selecting a sub-set of users for testing as above, but adding those users to the training data, in which case they will be the first rows (when passing ‘split_type=’joined’‘).

  This is meant to be used for fitting a model to all such training data, and then evaluating the produced user factors or top-K recommendations for the test users against the test data.

  It is recommended to use the ‘separated‘ mode, as it is more reflective of real scenarios, but some models or libraries do not have the capabilities for producing factors/recommendations for users which where not in the training data, and this option then comes in handy.

Usage

```
create.reco.train.test(
  X,
  split_type = "separated",
  users_test_fraction = 0.1,
  max_test_users = 10000L,
  items_test_fraction = 0.3,
  min_items_pool = 2L,
  min_pos_test = 1L,
  consider_cold_start = FALSE,
  seed = 1L
)
```

Arguments

`X` The implicit feedback data to split into training-testing-remainder for evaluating recommender systems. Should be passed as a sparse CSR matrix from the ‘Matrix‘ package (class ‘dgRMatrix‘). Users should correspond to rows, items to columns, and non-zero values to observed user-item interactions.

`split_type` Type of data split to generate. Allowed values are: ‘all‘, ‘separated‘, ‘joined‘ (see the function description above for more details).

`users_test_fraction`

Target fraction of the users to set as test (see the function documentation for more details). If the number represented by this fraction exceeds the number set by ‘max_test_users‘, then the actual number will be set to ‘max_test_users‘. Note however that the end result might end up containing fewer users if there are not enough users in the data meeting the minimum desired criteria.

If passing ‘NULL‘, will not take a fraction, but will instead take the number that is passed for ‘max_test_users‘.

Ignored when passing ‘split_type=’all’‘.

`max_test_users` Maximum number of users to set as test. Note that this will only be applied for choosing the minimum between this and ‘ncol(X)*users_test_fraction‘, while the actual number might end up being lower if there are not enough users meeting the desired minimum conditions.

If passing ‘NULL‘ for ‘users_test_fraction‘, will interpret this as the number of test users to take.

Ignored when passing ‘split_type=’all’‘.

`items_test_fraction`

Target fraction of the data (items) to set for test for each user. Should be a number between zero and one (non-inclusive). The actual number of test entries for each user will be determined as ‘round(n_entries_user*items_test_fraction)‘, thus in a long-tailed distribution (typical for recommender systems), the actual fraction that will be obtained is likely to be slightly lower than what is passed here.

Note that items are sampled independently for each user, thus the items that are in the test set for some users might be in the training set for different users.

`min_items_pool` Minimum number of items (sum of positive and negative items) that a user must have in order to be eligible as test user.

`min_pos_test` Minimum number of positive entries (non-zero entries in the test set) that users would need to have in order to be eligible as test user.

`consider_cold_start`

Whether to still set users as eligible for test in situations in which some user would have test data but no positive (non-zero) entries in the training data. This will only happen when passing ‘test_fraction>=0.5‘.

`seed` Seed to use for random number generation.

Value

Will return a list with two to four elements depending on the requested split type:

- If passing ‘split_type=’all’‘, will have elements ‘X_train‘ and ‘X_test‘, both of which will be sparse CSR matrices (class ‘dgRMatrix‘ from the ‘Matrix‘ package, which can be converted to other types through e.g. ‘MatrixExtra::as.csc.matrix‘) with the same number of rows and columns as the ‘X‘ that was passed as input.
- If passing ‘split_type=’separated’‘, will have the entries ‘X_train‘ and ‘X_test‘ as above (but with a number of rows corresponding to the number of selected test users instead), plus an entry ‘X_rem‘ which will be a CSR matrix containing the data for the remainder of the users (those which were not selected for testing and on which the recommendation model is meant to be fitted), and an entry ‘users_test‘ which will be an integer vector containing the indices of the users/rows in ‘X‘ which were selected for testing. The selected test users will be in sorted order, and the remaining data will remain in sorted order with the test users removed (e.g. if there’s 5 users, with the second and fifth selected for testing, then ‘X_train‘ and ‘X_test‘ will contain rows [2,5] of ‘X‘, while ‘X_rem‘ will contain rows [1,3,4]).
- If passing ‘split_type=’joined’‘, will not contain the entry ‘X_rem‘, but instead, ‘X_train‘ will be the concatenation of ‘X_train‘ and ‘X_rem‘, with ‘X_train‘ coming first (e.g. if there’s 5 users, with the second and fifth selected for testing, then ‘X_test‘ will contain rows [2,5] of ‘X‘, while ‘X_train‘ will contain rows [2,5,1,3,4], in that order).

The training and testing items for each user will not intersect, and each item in the original ‘X‘ data for a given test user will be assigned to either the training or the testing sets.

Index