RecTools

MTS Big Data

May 13, 2024

TABLE OF CONTENTS

RecTools is an easy-to-use Python library which makes the process of building recommendation systems easier, faster and more structured than ever before. The aim is to collect ready-to-use solutions and best practices in one place to make processes of creating your first MVP and deploying model to production as fast and easy as possible. The package also includes useful tools, such as ANN indexes for vector models and fast metric calculation.

CHAPTER

ONE

QUICK START

Download data.

```
$ wget https://files.grouplens.org/datasets/movielens/ml-1m.zip
$ unzip ml-1m.zip
```

Train model and infer recommendations.

```
import pandas as pd
from implicit.nearest_neighbours import TFIDFRecommender

from rectools import Columns
from rectools.dataset import Dataset
from rectools.models import ImplicitItemKNNWrapperModel

# Read the data
ratings = pd.read_csv(
    "ml-1m/ratings.dat",
    sep="::",
    engine="python",  # Because of 2-chars separators
    header=None,
    names=[Columns.User, Columns.Item, Columns.Weight, Columns.Datetime],
)

# Create dataset
dataset = Dataset.construct(ratings)

# Fit model
model = ImplicitItemKNNWrapperModel(TFIDFRecommender(K=10))
model.fit(dataset)

# Make recommendations
recos = model.recommend(
    users=ratings[Columns.User].unique(),
    dataset=dataset,
    k=10,
    filter_viewed=True,
)
```

CHAPTER

TWO

INSTALLATION

2.1 PyPI

Install from PyPi using pip

```
$ pip install rectools
```

RecTools is compatible with all operating systems and with Python 3.8+. The default version doesn't contain all the dependencies. Optional dependencies are the following:

lightfm: adds wrapper for LightFM model, torch: adds models based on neural nets, nmslib: adds fast ANN recommenders. all: all extra dependencies

Install RecTools with selected dependencies:

```
$ pip install rectools[lightfm,torch]
```

CHAPTER

THREE

WHY RECTOOLS?

The one, the only and the best.

RecTools provides unified interface for most commonly used recommender models. They include Implicit ALS, Implicit KNN, LightFM, SVD and DSSM. Recommendations based on popularity and random are also possible. For model validation, RecTools contains implementation of time split methodology and numerous metrics to evaluate model's performance. As well as basic ones they also include Diversity, Novelty and Serendipity. The package also provides tools that allow to evaluate metrics as easy and as fast as possible.

3.1 Components

3.1.1 Basic Concepts

Columns

Names of columns are fixed. They are *user_id*, *item_id*, *weight* (numerical value of interaction's importance), *datetime* (date and time of interaction), *rank* (rank of recommendation according to score) and *score* (numeric value estimating how good recommendation it is). Column names are fixed in order to not constantly require mapping of columns in data and their actual meaning. So you'll need to rename your columns.

`rectools.columns.Columns`()	Fixed column names for tables that contain interactions and recommendations.

Identifiers

Mappings of external identifiers of users or items to internal ones. Recommendation systems always require to have a mapping between external item ids in data sources and internal ids in interaction matrix. Managing such mapping requires a lot of diligence. RecTools does it for you. Every user and item must have a unique id. External ids may be any unique hashable values, internal - always integers from `0` to `n_objects-1`.

Interactions

This table stores history of interactions between users and items. It carries the most importance. Interactions table might also contain column describing importance of an interaction. Also timestamp of interaction. If no such column is provided, all interactions are assumed to be of equal importance.

User Features

This table stores data about users. It might include age, gender or any other features which may prove to be important for a recommender model.

Item Features

This table stores data about items. It might include category, price or any other features which may prove to be important for a recommender model.

Hot, warm, cold

There is a concept of a temperature we're using for users and items:

- **hot** - the ones that are present in interactions used for training (they may or may not have features);
- **warm** - the ones that are not in interactions, but have some features;
- **cold** - the ones we don't know anything about (they are not in interactions and don't have any features).

All the models are able to generate recommendations for the *hot* users (items). But as for warm and cold ones, there may be all possible combinations (neither of them, only cold, only warm, both). The important thing is that if model is able to recommend for cold users (items), but not for warm ones (see table below), it is still able to recommend for warm ones, but they will be considered as cold (no personalisation should be expected).

All of the above concepts are combined in *Dataset*. *Dataset* is used to build recommendation models and infer recommendations.

3.2 Dataset

3.2.1 Details of RecTools Dataset

See the API documentation for further details on Dataset:

rectools.dataset.dataset.Dataset(...[, ...])	Container class for all data for a recommendation model.
rectools.dataset.features.DenseFeatures(...)	Storage for dense features.
rectools.dataset.identifiers.IdMap(external_ids)	Mapping between external and internal object ids.
rectools.dataset.interactions.Interactions(df)	Structure to store info about user-item interactions.
rectools.dataset.features.SparseFeatures(...)	Storage for sparse features.

Dataset

class rectools.dataset.dataset.**Dataset**(*user_id_map: IdMap*, *item_id_map: IdMap*, *interactions: Interactions*, *user_features: Optional[Union[DenseFeatures, SparseFeatures]] = None*, *item_features: Optional[Union[DenseFeatures, SparseFeatures]] = None*)

Bases: `object`

Container class for all data for a recommendation model.

It stores data about internal-external id mapping, user-item interactions, user and item features in special *rectools* structures for convenient future usage.

WARNING: It's highly not recommended to create *Dataset* object directly. Use *construct* class method instead.

Parameters

- **user_id_map** (IdMap) – User identifiers mapping.
- **item_id_map** (IdMap) – Item identifiers mapping.
- **interactions** (Interactions) – User-item interactions.
- **user_features** (DenseFeatures *or* SparseFeatures, *optional*) – User explicit features.
- **item_features** (DenseFeatures *or* SparseFeatures, *optional*) – Item explicit features.

Inherited-members

Methods

construct(interactions_df[, ...])	Class method for convenient *Dataset* creation.
get_hot_item_features()	Item features for hot items.
get_hot_user_features()	User features for hot users.
get_raw_interactions([include_weight, ...])	Return interactions as a *pd.DataFrame* object with replacing internal user and item ids to external ones.
get_user_item_matrix([include_weights, ...])	Construct user-item CSR matrix based on *interactions* attribute.

Attributes

user_id_map
item_id_map
interactions
user_features
item_features

n_hot_items	Return number of hot items in dataset.
n_hot_users	Return number of hot users in dataset.

classmethod construct(*interactions_df: DataFrame*, *user_features_df: Optional[DataFrame] = None*, *cat_user_features: Iterable[str] = ()*, *make_dense_user_features: bool = False*, *item_features_df: Optional[DataFrame] = None*, *cat_item_features: Iterable[str] = ()*, *make_dense_item_features: bool = False*) → *Dataset*

Class method for convenient *Dataset* creation.

Use it to create dataset from raw data.

Parameters

- **interactions_df** (*pd.DataFrame*) –

  Table where every row contains user-item interaction and columns are:

  - *Columns.User* - user id;
  - *Columns.Item* - item id;
  - *Columns.Weight* - weight of interaction, *float*, use 1 if interactions have no weight;
  - *Columns.Datetime* - timestamp of interactions, assign random value if you're not going to use it later.

- **user_features_df** (*pd.DataFrame, optional*) – User (item) explicit features table. It will be used to create *SparseFeatures* using *from_flatten* class method or *DenseFeatures* using *from_dataframe* class method depending on *make_dense_user_features* (*make_dense_item_features*) flag. See detailed info about the table structure in these methods description.
- **item_features_df** (*pd.DataFrame, optional*) – User (item) explicit features table. It will be used to create *SparseFeatures* using *from_flatten* class method or *DenseFeatures* using *from_dataframe* class method depending on *make_dense_user_features* (*make_dense_item_features*) flag. See detailed info about the table structure in these methods description.
- **cat_user_features** (tp.Iterable[str], default `()`) – List of categorical user (item) feature names for *SparseFeatures.from_flatten* method. Used only if *make_dense_user_features* (*make_dense_item_features*) flag is `False` and *user_features_df* (*item_features_df*) is not `None`.
- **cat_item_features** (tp.Iterable[str], default `()`) – List of categorical user (item) feature names for *SparseFeatures.from_flatten* method. Used only if *make_dense_user_features* (*make_dense_item_features*) flag is `False` and *user_features_df* (*item_features_df*) is not `None`.
- **make_dense_user_features** (bool, default `False`) – Create user (item) features as dense or sparse. Used only if *user_features_df* (*item_features_df*) is not `None`. - if `False`, *SparseFeatures.from_flatten* method will be used; - if `True`, *DenseFeatures.from_dataframe* method will be used.
- **make_dense_item_features** (bool, default `False`) – Create user (item) features as dense or sparse. Used only if *user_features_df* (*item_features_df*) is not `None`. - if `False`, *SparseFeatures.from_flatten* method will be used; - if `True`, *DenseFeatures.from_dataframe* method will be used.

Returns

Container with all input data, converted to *rectools* structures.

Return type

Dataset

`get_hot_item_features`() → Optional[Union[*DenseFeatures*, *SparseFeatures*]]

Item features for hot items.

Return type
Optional[*Union*[DenseFeatures, SparseFeatures]]

`get_hot_user_features`() → Optional[Union[*DenseFeatures*, *SparseFeatures*]]

User features for hot users.

Return type
Optional[*Union*[DenseFeatures, SparseFeatures]]

`get_raw_interactions`(*include_weight: bool = True*, *include_datetime: bool = True*) → DataFrame

Return interactions as a *pd.DataFrame* object with replacing internal user and item ids to external ones.

Parameters

- **include_weight** (bool, default `True`) – Whether to include weight column into resulting table or not.
- **include_datetime** (bool, default `True`) – Whether to include datetime column into resulting table or not.

Return type
pd.DataFrame

`get_user_item_matrix`(*include_weights: bool = True*, *include_warm_users: bool = False*, *include_warm_items: bool = False*) → csr_matrix

Construct user-item CSR matrix based on *interactions* attribute.

Return a resized user-item matrix. Resizing is done using *user_id_map* and *item_id_map*, hence if either a user or an item is not presented in interactions, but presented in id map, then it's going to be in the returned matrix.

Parameters

- **include_weights** (bool, default `True`) – Whether include interaction weights in matrix or not. If False, all values in returned matrix will be equal to 1.
- **include_warm** (bool, default `False`) – Whether to include warm users and items into the matrix or not. Rows and columns for warm users and items will be added to the end of matrix, they will contain only zeros.
- **include_warm_users** (*bool*) –
- **include_warm_items** (*bool*) –

Returns
Resized user-item CSR matrix

Return type
csr_matrix

`property n_hot_items: int`

Return number of hot items in dataset. Items with internal ids from *0* to *n_hot_items - 1* are hot (they are present in interactions). Items with internal ids from *n_hot_items* to *dataset.item_id_map.size - 1* are warm (they aren't present in interactions, but they have features).

`property n_hot_users: int`

Return number of hot users in dataset. Users with internal ids from *0* to *n_hot_users - 1* are hot (they are present in interactions). Users with internal ids from *n_hot_users* to *dataset.user_id_map.size - 1* are warm (they aren't present in interactions, but they have features).

DenseFeatures

class rectools.dataset.features.**DenseFeatures**(*values: ndarray*, *names: Tuple[str, ...]*)

Bases: object

Storage for dense features.

Dense features are represented as a dense matrix, where rows correspond to objects, columns - to features.

Usually you do not need to create this object directly, use *from_dataframe* class method instead. If you want to use custom logic, use *from_iterables* class method instead of direct creation.

Parameters

- **values** (*np.ndarray*) – Matrix of feature values in the classic format: rows - objects, columns - features.
- **names** (*tuple(str)*) – Names of features (number of names must be equal to the number of columns in values).

Inherited-members

Methods

from_dataframe(df, id_map[, id_col])	Create DenseFeatures object from dataframe.
from_iterables(values, names)	Create class instance from any iterables of feature values and names.
get_dense()	Return values in dense format.
get_sparse()	Return values in sparse format.
take(ids)	Take a subset of features for given subject (user or item) ids.

Attributes

values
names

classmethod **from_dataframe**(*df: DataFrame*, *id_map:* IdMap, *id_col: str = 'id'*) → *DenseFeatures*

Create DenseFeatures object from dataframe.

Assume that feature values are values in dataframe, and feature names are column names.

Parameters

- **df** (*pd.Dataframe*) – Table in classic format: rows corresponds to objects, columns - to features. One special column *id_col* must contain object external ids.
- **id_map** (IdMap) – Mapping between external and internal ids. Sets of ids in *id_map* and in *df* must be equal.
- **id_col** (str, default id) – Name of column containing object ids.

Return type

DenseFeatures

classmethod from_iterables(*values: Iterable[Iterable[float]]*, *names: Iterable[str]*) → *DenseFeatures*

Create class instance from any iterables of feature values and names.

Parameters

- **values** (*iterable(iterable(float))*) – Feature values matrix. E.g. list of lists: [[1, 2, 3], [4, 5, 6]].
- **names** (*iterable(str)*) – Feature names.

Return type
DenseFeatures

get_dense() → ndarray

Return values in dense format.

Return type
ndarray

get_sparse() → csr_matrix

Return values in sparse format.

Return type
csr_matrix

take(*ids: Union[Sequence[int], ndarray]*) → *DenseFeatures*

Take a subset of features for given subject (user or item) ids.

Parameters
ids (*array-like*) – Array of internal ids to select features for.

Return type
DenseFeatures

IdMap

class rectools.dataset.identifiers.**IdMap**(*external_ids: ndarray*)

Bases: `object`

Mapping between external and internal object ids.

External ids may be any unique hashable values, internal - always integers from `0` to `n_objects-1`.

Usually you do not need to create this object directly, use *from_values* class method instead.

When creating directly you have to pass unique external ids.

Parameters
external_ids (*np.ndarray*) – Array of *unique* external ids.

Inherited-members

Methods

add_ids(values[, raise_if_already_present])	Add new external ids to current IdMap and return new IdMap.
convert_to_external()	Convert any sequence of internal ids to array of external ids (map internal -> external).
convert_to_internal()	Convert any sequence of external ids to array of internal ids (map external -> internal).
from_dict(mapping)	Create IdMap from dict of external id -> internal id mappings.
from_values(values)	Create IdMap from list of external ids (possibly not unique).
get_external_sorted_by_internal()	Return array of external ids sorted by internal ids.
get_sorted_internal()	Return array of sorted internal ids.

Attributes

external_ids

external_dtype	Return dtype of external ids.
internal_ids	Array of internal ids.
size	Return number of ids in map.
to_external	Map internal->external.
to_internal	Map internal->external.

add_ids(*values: Union[Sequence[Hashable], ndarray]*, *raise_if_already_present: bool = False*) → *IdMap*

Add new external ids to current IdMap and return new IdMap. Mapping for old ids does not change. New ids are added to the end of list of external ids.

Parameters

- **values** (*iterable(hashable)*) – List of new external ids (may be not unique).
- **raise_if_already_present** (bool, default False) – If True and some of given ids are already present in the map ValueError will be raised.

Return type
IdMap

Raises
ValueError – If some of given ids are already present in the map and *raise_if_already_present* flag is True.

convert_to_external(*internal: Union[Sequence[int], ndarray]*, *strict: bool = True*, *return_missing: Literal[False] = False*) → ndarray

convert_to_external(*internal: Union[Sequence[int], ndarray]*, *strict: bool = True*, ***, *return_missing: Literal[True]*) → Tuple[ndarray, ndarray]

Convert any sequence of internal ids to array of external ids (map internal -> external).

Parameters

- **internal** (*sequence(int)*) – Sequence of internal ids to convert.
- **strict** (bool, default True) –

Defines behaviour when some of given internal ids do not exist in mapping.

- If `True`, *KeyError* will be raised;
- If `False`, nonexistent ids will be skipped.

- **return_missing** (bool, default `False`) – If True, return a tuple of 2 arrays: external ids and missing ids (that are not in map). Works only if *strict* is False.

Returns

- *np.ndarray* – Array of external ids.
- *np.ndarray, np.ndarray* – Tuple of 2 arrays: external ids and missing ids. Only if *strict* is False and *return_missing* is True.

Raises

- **KeyError** – If some of given internal ids do not exist in mapping and *strict* flag is True.
- **ValueError** – If *strict* and *return_missing* are both `True`.

convert_to_internal(*external: Union[Sequence[Hashable], ndarray]*, *strict: bool = True*, *return_missing: Literal[False] = False*) → ndarray

convert_to_internal(*external: Union[Sequence[Hashable], ndarray]*, *strict: bool = True*, ***, *return_missing: Literal[True]*) → Tuple[ndarray, ndarray]

Convert any sequence of external ids to array of internal ids (map external -> internal).

Parameters

- **external** (*sequence(hashable)*) – Sequence of external ids to convert.
- **strict** (bool, default `True`) –

  Defines behaviour when some of given external ids do not exist in mapping.

  - If `True`, *KeyError* will be raised;
  - If `False`, nonexistent ids will be skipped.

- **return_missing** (bool, default `False`) – If True, return a tuple of 2 arrays: internal ids and missing ids (that are not in map). Works only if *strict* is False.

Returns

- *np.ndarray* – Array of internal ids.
- *np.ndarray, np.ndarray* – Tuple of 2 arrays: internal ids and missing ids. Only if *strict* is False and *return_missing* is True.

Raises

- **KeyError** – If some of given external ids do not exist in mapping and *strict* flag is `True`.
- **ValueError** – If *strict* and *return_missing* are both `True`.

property external_dtype: Type

Return dtype of external ids.

classmethod from_dict(*mapping: Dict[Hashable, int]*) → *IdMap*

Create IdMap from dict of external id -> internal id mappings. Could be used if mappings were previously defined somewhere else.

Parameters
mapping (*dict(hashable, int) :*) – Dict of mappings from external ids to internal ids. Internal ids must be integers from 0 to n_objects-1.

Return type
IdMap

classmethod from_values(*values: Union[Sequence[Hashable], ndarray]*) → *IdMap*

Create IdMap from list of external ids (possibly not unique).

Parameters
values (*iterable(hashable) :*) – List of all external ids (may be not unique).

Return type
IdMap

get_external_sorted_by_internal() → ndarray

Return array of external ids sorted by internal ids.

Return type
ndarray

get_sorted_internal() → ndarray

Return array of sorted internal ids.

Return type
ndarray

property internal_ids: ndarray

Array of internal ids.

property size: int

Return number of ids in map.

property to_external: Series

Map internal->external.

property to_internal: Series

Map internal->external.

Interactions

class rectools.dataset.interactions.**Interactions**(*df: DataFrame*)

Bases: object

Structure to store info about user-item interactions.

Usually it's more convenient to use *from_raw* method instead of direct creating.

Parameters
df (*pd.DataFrame*) –

Table where every row contains user-item interaction and columns are:

- *Columns.User* - internal user id (non-negative int values);
- *Columns.Item* - internal item id (non-negative int values);
- *Columns.Weight* - weight of interaction, float, use 1 if interactions have no weight;
- *Columns.Datetime* - timestamp of interactions, assign random value if you're not going to use it later.

Inherited-members

Methods

from_raw(interactions, user_id_map, item_id_map)	Create *Interactions* from dataset with external ids and id mappings.
get_user_item_matrix([include_weights])	Form a user-item CSR matrix based on interactions data.
to_external(user_id_map, item_id_map[, ...])	Convert itself to *pd.DataFrame* with replacing internal user and item ids to external ones.

Attributes

df

classmethod from_raw(*interactions: DataFrame*, *user_id_map:* IdMap, *item_id_map:* IdMap) → *Interactions*

Create *Interactions* from dataset with external ids and id mappings.

Parameters

- **interactions** (*pd.DataFrame*) –

  Table where every row contains user-item interaction and columns are:

  - *Columns.User* - user id;
  - *Columns.Item* - item id;
  - *Columns.Weight* - weight of interaction, float, use 1 if interactions have no weight;
  - *Columns.Datetime* - timestamp of interactions, assign random value if you're not going to use it later.

- **user_id_map** (IdMap) – User identifiers mapping.
- **item_id_map** (IdMap) – Item identifiers mapping.

Return type
Interactions

get_user_item_matrix(*include_weights: bool = True*) → csr_matrix

Form a user-item CSR matrix based on interactions data.

Parameters
include_weights (bool, default True) – Whether include interaction weights in matrix or not. If False, all values in returned matrix will be equal to 1.

Return type
csr_matrix

to_external(*user_id_map:* IdMap, *item_id_map:* IdMap, *include_weight: bool = True*, *include_datetime: bool = True*) → DataFrame

Convert itself to *pd.DataFrame* with replacing internal user and item ids to external ones.

Parameters

- **user_id_map** (IdMap) – User id map that has to be used for converting internal user ids to external ones.
- **item_id_map** (IdMap) – Item id map that has to be used for converting internal item ids to external ones.
- **include_weight** (bool, default True) – Whether to include weight column into resulting table or not
- **include_datetime** (bool, default True) – Whether to include datetime column into resulting table or not.

Return type
pd.DataFrame

SparseFeatures

class rectools.dataset.features.**SparseFeatures**(*values: csr_matrix*, *names: Tuple[Tuple[str, Any], ...]*)

Bases: object

Storage for sparse features.

Sparse features are represented as CSR matrix, where rows correspond to objects, columns - to features. Assume that there are features of two types: direct and categorical.

Each direct feature is represented in a single column with its real values. Direct features are numeric. E.g. +—+—-+—-+ | | f1 | f2 | +—+—+—-+—-+ | 1 | 23 | 3 | +—+—+—-+—-+ | 2 | 36 | 5 | +—+—+—-+—-+

Categorical features are one-hot encoded (https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html), values in matrix are counts in category. If you want to binarize a numeric feature, make it categorical with bin indices as categories. E.g. +—+——+——+——+ | | f1_a | f1_b | f2_1 | +—+——+——+——+ | 1 | 0 | 2 | 1 | +—+——+——+——+ | 2 | 1 | 1 | 0 | +—+——+——+——+

Usually you do not need to create this object directly, use *from_flatten* class method instead. If you want to use custom logic, use *from_iterables* class method instead of direct creation.

Parameters

- **values** (*csr_matrix*) – CSR matrix containing OHE feature values.
- **names** (*tuple(tuple(str, any))*) – Tuple of feature names. Direct features are represented only by names, so for direct features use (feature name, *None*). For sparse features use (feature name, value), as they are one-hot encoded. E.g. If you have direct feature *age* and cat. feature *sex*, names will be *((age, None), (sex, m), (sex, f))*. Number of names must be equal to the number of columns in values.

Inherited-members

Methods

from_flatten(df, id_map[, cat_features, ...])	Construct *SparseFeatures* from flatten DataFrame.
from_iterables(values, names)	Create class instance from sparse matrix and iterable feature names.
get_dense()	Return values in dense format.
get_sparse()	Return values in sparse format.
take(ids)	Take a subset of features for given subject (user or item) ids.

Attributes

values
names

classmethod from_flatten(*df: DataFrame*, *id_map:* IdMap, *cat_features: Iterable[Any] = ()*, *id_col: str = 'id'*, *feature_col: str = 'feature'*, *value_col: str = 'value'*, *weight_col: str = 'weight'*) → *SparseFeatures*

Construct *SparseFeatures* from flatten DataFrame.

Flatten DataFrame has 3 obligatory columns: <id of object>, <feature name>, <feature value>, and <feature weight> as the optional fourth. If there is no <feature weight> column, all weights will be assumed to be equal to 1.

Direct features converted to sparse matrix as is. Its values are multiplied by weights. Values for the same object and same feature are added up. E.g: +—-+———+———-+———+ | id | feature | value | weight | +—-+———+———-+———+ | 1 | f1 | 10 | 1 | +—-+———+———-+———+ | 2 | f1 | 20 | 1.5 | +—-+———+———-+———+ | 1 | f1 | 15 | 1 | +—-+———+———-+———+ | 2 | f2 | 3 | 1 | +—-+———+———-+———+ Out: +—+—-+—-+—-+ | | f1 | f2 | +—+—-+—-+—-+ | 1 | 25 | | +—+—-+—-+—-+ | 2 | 30 | 3 | +—+—-+—-+—-+

Categorical features are represented as horizontally stacked one-hot vectors. Duplicated values are counted. Final values (counts) are multiplied by weights. E.g: +—-+———+———-+———+ | id | feature | value | weight | +—-+———+———-+———+ | 1 | f1 | 10 | 1 | +—-+———+———-+———+ | 2 | f1 | 20 | 1.5 | +—-+———+———-+———+ | 1 | f1 | 10 | 1 | +—-+———+———-+———+ | 2 | f2 | 3 | 1 | +—-+———+———-+———+

Out: +—+———+———+———-+ | | f1__10 | f1__20 | f2__3 | +—+———+———+———-+ | 1 | 2 | | | +—+———+———+———-+ | 2 | | 1.5 | 1 | +—+———+———+———-+

Parameters

- **df** (*pd.DataFrame*) – Flatten table with features with columns *id_col*, *feature_col*, *value_col* in format described above.
- **id_map** (IdMap) – Mapping between external and internal ids.
- **cat_features** (iterable(str), default ()) – List of categorical feature names.
- **id_col** (str, default id) – Name of column with object ids.
- **feature_col** (str, default feature) – Name of column with feature names.
- **value_col** (str, default value) – Name of column with feature values.

- **weight_col** (str, default `weight`) – Name of column with feature weight. If no such column provided, all weights will be equal to 1.

Return type
SparseFeatures

classmethod from_iterables(*values: csr_matrix*, *names: Iterable[Tuple[str, Any]]*) → *SparseFeatures*

Create class instance from sparse matrix and iterable feature names.

Parameters

- **values** (*csr_matrix*) – Feature values matrix.
- **names** (*iterable((str, any))*) – Feature names in same format as in constructor.

Return type
SparseFeatures

get_dense() → ndarray

Return values in dense format.

Return type
ndarray

get_sparse() → csr_matrix

Return values in sparse format.

Return type
csr_matrix

take(*ids: Union[Sequence[int], ndarray]*) → *SparseFeatures*

Take a subset of features for given subject (user or item) ids.

Parameters
ids (*array-like*) – Array of internal ids to select features for.

Return type
SparseFeatures

3.3 Models

3.3.1 Details of RecTools Models

Model	Supports features	Recommends for warm	Recommends for cold
DSSMModel	Yes	Yes	No
EASEModel	No	No	No
ImplicitALSWrapperModel	Yes	No	No
ImplicitItemKNNWrapperModel	No	No	No
LightFMWrapperModel	Yes	Yes	Yes
PopularModel	No	No	Yes
PopularInCategoryModel	No	No	Yes
PureSVDModel	No	No	No
RandomModel	No	No	Yes

See the API documentation for further details on Models:

rectools.models.dssm.DSSMModel(...)	Wrapper for *rectools.models.dssm.DSSM*
rectools.models.ease.EASEModel([...])	Embarrassingly Shallow Autoencoders for Sparse Data model.
rectools.models.implicit_als.ImplicitALSWrapperModel(model)	Wrapper for *implicit.als.AlternatingLeastSquares* with possibility to use explicit features and GPU support.
rectools.models.implicit_knn.ImplicitItemKNNWrapperModel(model)	Wrapper for *implicit.nearest_neighbours.ItemItemRecommender* and its successors.
rectools.models.lightfm.LightFMWrapperModel(model)	Wrapper for *lightfm.LightFM*.
rectools.models.popular_in_category.PopularInCategoryModel(...)	Model generating recommendations based on popularity of items.
rectools.models.popular.PopularModel([...])	Model generating recommendations based on popularity of items.
rectools.models.pure_svd.PureSVDModel([...])	PureSVD matrix factorization model.
rectools.models.random.RandomModel([...])	Model generating random recommendations.

DSSMModel

class rectools.models.dssm.**DSSMModel**(*train_dataset_type: ~typing.Type[~rectools.dataset.torch_datasets.DSSMTrainDatasetBase] = <class 'rectools.dataset.torch_datasets.DSSMTrainDataset'>, user_dataset_type: ~typing.Type[~rectools.dataset.torch_datasets.DSSMUserDatasetBase] = <class 'rectools.dataset.torch_datasets.DSSMUserDataset'>, item_dataset_type: ~typing.Type[~rectools.dataset.torch_datasets.DSSMItemDatasetBase] = <class 'rectools.dataset.torch_datasets.DSSMItemDataset'>, model: ~typing.Optional[~rectools.models.dssm.DSSM] = None, n_factors: int = 128, max_epochs: int = 5, batch_size: int = 128, dataloader_num_workers: int = 0, trainer_sanity_steps: int = 2, trainer_devices: ~typing.Union[str, int] = 1, trainer_accelerator: str = 'auto', callbacks: ~typing.Optional[~typing.Union[~typing.List[~pytorch_lightning.callbacks.callback.Callback], ~pytorch_lightning.callbacks.callback.Callback]] = None, loggers: ~typing.Union[~pytorch_lightning.loggers.logger.Logger, ~typing.Iterable[~pytorch_lightning.loggers.logger.Logger], bool] = True, verbose: int = 0, deterministic: bool = False*)

Bases: VectorModel

Wrapper for *rectools.models.dssm.DSSM*

Parameters

- **train_dataset_type** (Type(DSSMTrainDatasetBase), default *DSSMTrainDataset*) – Type of dataset used for training. A child of *torch.utils.data.Dataset* that implements *from_dataset* classmethod. Used to construct *torch.utils.data.Dataset* from a given *rectools.dataset.dataset.Dataset*.
- **user_dataset_type** (Type(DSSMUserDatasetBase), default *DSSMUserDataset*) – Type of dataset used for user inference. A child of *torch.utils.data.Dataset* that implements *from_dataset* classmethod. Used to construct *torch.utils.data.Dataset* from a given *rectools.dataset.dataset.Dataset*.
- **item_dataset_type** (Type(DSSMItemDatasetBase), default *DSSMItemDataset*) –

Type of dataset used for item inference. A child of *torch.utils.data.Dataset* that implements *from_dataset* classmethod. Used to construct *torch.utils.data.Dataset* from a given *rectools.dataset.dataset.Dataset*.

- **model** (*Optional(DSSM), default None*) – Which model to wrap. If model is None, an instance of default DSSM is created during fit.
- **n_factors** (*int, default 128*) – How many hidden units to use in user and item networks. Used only if *model* is None.
- **max_epochs** (*int, default 5*) – Stop training if this number of epochs is reached. Keep in mind that if any kind of early stopping callback is passed as one of the callbacks along with a validation dataset, then hitting exactly max_epochs is not guaranteed.
- **batch_size** (*int, default 128*) – How many samples per batch to load.
- **dataloader_num_workers** (*int, default 0*) – How many processes to use for data loading. Defaults to 0, which means that all data will be loaded in the main process.
- **trainer_sanity_steps** (*int, default 2*) – Sanity check runs n validation batches before starting the training routine.
- **trainer_devices** (*str | int, default 1*) – "auto" means determine the number of available devices based on the *trainer_accelerator* type. In case on an integer, it will be mapped to either *gpus*, *tpu_cores*, *num_processes* or *ipus*, based on the accelerator type.
- **trainer_accelerator** (*str, default 'auto'*) – Supports passing different accelerator types ("cpu", "gpu", "tpu", "ipu", "auto"). The "auto" option recognizes the machine you are on, and selects the respective.
- **callbacks** (*Optional(Sequence(Callback)), default None*) – Which callbacks to use. For instance, *pytorch_lightning.callbacks.TQDMProgressBar*, etc.
- **loggers** (*LightningLoggerBase | iterable(LightningLoggerBase) | bool, default True*) – Which loggers to use. For instance, *pytorch_lightning.loggers.TensorboardLogger*, etc.
- **verbose** (*int, default 0*) – Verbosity level (applies only to recommend loop).
- **deterministic** (bool, default `False`) – If `True`, sets whether PyTorch operations must use deterministic algorithms. Use *pytorch_lightning.seed_everything* together with this param to fix the random state.

Inherited-members

Methods

`fit`(dataset, *args, **kwargs)	Fit model.
`get_vectors`(dataset)	

`recommend`(users, dataset, k, filter_viewed)	Recommend items for users.
`recommend_to_items`(target_items, dataset, k)	Recommend items for target items.

Attributes

i2i_dist
n_threads
recommends_for_cold
recommends_for_warm
u2i_dist

EASEModel

class rectools.models.ease.**EASEModel**(*regularization: float = 500.0*, *num_threads: int = 1*, *verbose: int = 0*)

Bases: *ModelBase*

Embarrassingly Shallow Autoencoders for Sparse Data model.

See https://arxiv.org/abs/1905.03375.

Please note that this algorithm requires a lot of RAM during *fit* method. Out-of-memory issues are possible for big datasets. Reasonable catalog size for local development is about 30k items. Reasonable amount of interactions is about 20m.

Parameters

- **regularization** (*float*) – The regularization factor of the weights.
- **verbose** (*int, default 0*) – Degree of verbose output. If 0, no output will be provided.
- **num_threads** (*int, default 1*) – Number of threads used for *recommend* method.

Inherited-members

Methods

fit(dataset, *args, **kwargs)	Fit model.
recommend(users, dataset, k, filter_viewed)	Recommend items for users.
recommend_to_items(target_items, dataset, k)	Recommend items for target items.

Attributes

recommends_for_cold
recommends_for_warm

ImplicitALSWrapperModel

class rectools.models.implicit_als.**ImplicitALSWrapperModel**(*model: Union[AlternatingLeastSquares, AlternatingLeastSquares]*, *verbose: int = 0*, *fit_features_together: bool = False*)

Bases: *VectorModel*

Wrapper for *implicit.als.AlternatingLeastSquares* with possibility to use explicit features and GPU support.

See https://implicit.readthedocs.io/en/latest/als.html for details of base model.

Parameters

- **model** (*AnyAlternatingLeastSquares*) – Base model that will be used.
- **verbose** (*int, default 0*) – Degree of verbose output. If 0, no output will be provided.
- **fit_features_together** (*bool, default False*) – Whether fit explicit features together with latent features or not. Used only if explicit features are present in dataset. See documentations linked above for details.

Inherited-members

Methods

fit(dataset, *args, **kwargs)	Fit model.
get_vectors()	Return user and item vector representations from fitted model.
recommend(users, dataset, k, filter_viewed)	Recommend items for users.
recommend_to_items(target_items, dataset, k)	Recommend items for target items.

Attributes

i2i_dist
n_threads
recommends_for_cold
recommends_for_warm
u2i_dist

get_vectors() → Tuple[ndarray, ndarray]

Return user and item vector representations from fitted model.

Returns
User and item embeddings. Shapes are (n_users, n_factors) and (n_items, n_factors).

Return type
(np.ndarray, np.ndarray)

ImplicitItemKNNWrapperModel

class rectools.models.implicit_knn.**ImplicitItemKNNWrapperModel**(*model: ItemItemRecommender*, *verbose: int = 0*)

Bases: *ModelBase*

Wrapper for *implicit.nearest_neighbours.ItemItemRecommender* and its successors.

See https://github.com/benfred/implicit/blob/main/implicit/nearest_neighbours.py for details.

Parameters

- **model** (*ItemItemRecommender*) – Base model that will be used.
- **verbose** (*int, default 0*) – Degree of verbose output. If 0, no output will be provided.

Inherited-members

Methods

fit(dataset, *args, **kwargs)	Fit model.
recommend(users, dataset, k, filter_viewed)	Recommend items for users.
recommend_to_items(target_items, dataset, k)	Recommend items for target items.

Attributes

recommends_for_cold
recommends_for_warm

LightFMWrapperModel

class rectools.models.lightfm.**LightFMWrapperModel**(*model: LightFM*, *epochs: int = 1*, *num_threads: int = 1*, *verbose: int = 0*)

Bases: *FixedColdRecoModelMixin*, *VectorModel*

Wrapper for *lightfm.LightFM*.

See https://making.lyst.com/lightfm/docs/home.html for details of base model.

SparseFeatures are used for this model, if you use DenseFeatures, it'll be converted to sparse. Also it's usually better to use categorical features. If you have real features (age, price, etc.), you can binarize it.

Parameters

- **model** (*LightFM*) – Base model that will be used.
- **epochs** (*int, default 1*) – Will be used as *epochs* parameter for *LightFM.fit*.
- **num_threads** (*int, default 1*) – Will be used as *num_threads* parameter for *LightFM.fit*.
- **verbose** (*int, default 0*) – Degree of verbose output. If 0, no output will be provided.

Inherited-members

Methods

fit(dataset, *args, **kwargs)	Fit model.
get_vectors(dataset[, add_biases])	Return user and item vector representations from fitted model.
recommend(users, dataset, k, filter_viewed)	Recommend items for users.
recommend_to_items(target_items, dataset, k)	Recommend items for target items.

Attributes

i2i_dist
n_threads
recommends_for_cold
recommends_for_warm
u2i_dist

get_vectors(*dataset: Dataset*, *add_biases: bool = True*) → Tuple[ndarray, ndarray]

Return user and item vector representations from fitted model.

Parameters

- **dataset** (*Dataset*) – Dataset with input data. Usually it's the same dataset that was used to fit model.
- **add_biases** (*bool, default True*) – LightFM model stores separately embeddings and biases for users and items. If *False*, only embeddings will be returned. If *True*, biases will be added as 2 first columns (see *Returns* section for details).

Returns

User and item embeddings.

If *add_biases* is `False`, shapes are `(n_users, no_components)` and `(n_items, no_components)`.

If *add_biases* is `True`, shapes are `(n_users, no_components + 2)` and `(n_items, no_components + 2)`. In that case `(user_biases_column, ones_column)` will be added to user embeddings, and `(ones_column, item_biases_column)` - to item embeddings. So, if you calculate *user_embeddings @ item_embeddings.T*, for each user-item pair you will get value *user_embedding @ item_embedding + user_bias + item_bias*.

Return type

(np.ndarray, np.ndarray)

PopularInCategoryModel

class rectools.models.popular_in_category.**PopularInCategoryModel**(*category_feature: str*, *n_categories: Optional[int] = None*, *mixing_strategy: Optional[str] = 'rotate'*, *ratio_strategy: Optional[str] = 'proportional'*, *popularity: str = 'n_users'*, *period: Optional[timedelta] = None*, *begin_from: Optional[datetime] = None*, *add_cold: bool = False*, *inverse: bool = False*, *verbose: int = 0*)

Bases: *PopularModel*

Model generating recommendations based on popularity of items.

Parameters

- **category_feature** (*str*) – Name of category feature in item features dataframe.
- **n_categories** (int, optional, default None) – Number of most popular categories to take for recommendations
- **mixing_strategy** ({"rotate", "group"}, default *"rotate"*) – Method of mixing recommendations from different categories. The following methods are available: - *rotate* - items from different categories take turns in final recommendations, starting from the most popular category - *group* - items from each category are grouped together. Categories are sorted by popularity
- **ratio_strategy** ({"equal", "proportional"}, default *"proportional"*) – Method of defining ratios for categories. The following methods are available: - *equal* - all categories gain equal ratios in recommendations. Exceeding places for items are given to most popular categories - *proportional* - categories gain ratios in recommendations based on their popularity. Each category gains at least one item in recommendations if number of categories doesn't exceed number of recs.
- **popularity** ({"n_users", "n_interactions", "mean_weight", "sum_weight"}, default *"n_users"*) – Method of calculating item popularity. To evaluate *popularity score* the following methods are available: - *n_users* - number of unique users that interacted with item; - *n_interactions* - number of interactions with item; - *mean_weight* - mean item interactions weight; - *sum_weight* - total item interactions weight.
- **period** (timedelta, optional, default None) – Period before last interaction to consider interactions for popularity calculation. Either *period* or *begin_from* can be set at once. If both are None all interactions will be used.
- **begin_from** (datetime, optional, default None) – Exact datetime to consider interactions from for popularity calculation. Either *period* or *begin_from* can be set at once. If both are None all interactions will be used.
- **add_cold** (bool, default False) – If True cold items will be added to the end of popularity list and can be recommended. Item is cold if it's not present in interactions at all (but present in id map) or not present in last interactions defined by either *period* or *begin_from* arguments. Order of cold items is unpredictable. Cold items score will be equal to 0.
- **inverse** (bool, default False) – If True least popular items will be selected.
- **verbose** (int, default 0) – Degree of verbose output. If 0, no output will be provided.

Inherited-members

Methods

`fit`(dataset, *args, **kwargs)	Fit model.
`recommend`(users, dataset, k, filter_viewed)	Recommend items for users.
`recommend_to_items`(target_items, dataset, k)	Recommend items for target items.

Attributes

`recommends_for_cold`
`recommends_for_warm`

PopularModel

class rectools.models.popular.**PopularModel**(*popularity: str = 'n_users'*, *period: Optional[timedelta] = None*, *begin_from: Optional[datetime] = None*, *add_cold: bool = False*, *inverse: bool = False*, *verbose: int = 0*)

Bases: *FixedColdRecoModelMixin*, *ModelBase*

Model generating recommendations based on popularity of items.

Parameters

- **popularity** ({"n_users", "n_interactions", "mean_weight", "sum_weight"}, default *"n_users"*) – Method of calculating item popularity. To evaluate *popularity score* the following methods are available: - *n_users* - number of unique users that interacted with item; - *n_interactions* - number of interactions with item; - *mean_weight* - mean item interactions weight; - *sum_weight* - total item interactions weight.
- **period** (timedelta, optional, default `None`) – Period before last interaction to consider interactions for popularity calculation. Either *period* or *begin_from* can be set at once. If both are `None` all interactions will be used.
- **begin_from** (datetime, optional, default `None`) – Exact datetime to consider interactions from for popularity calculation. Either *period* or *begin_from* can be set at once. If both are `None` all interactions will be used.
- **add_cold** (bool, default `False`) – If `True` cold items will be added to the end of popularity list and can be recommended. Item is cold if it's not present in interactions at all (but present in id map) or not present in last interactions defined by either *period* or *begin_from* arguments. Order of cold items is unpredictable. Cold items score will be equal to `0`.
- **inverse** (bool, default `False`) – If `True` least popular items will be selected.
- **verbose** (int, default `0`) – Degree of verbose output. If `0`, no output will be provided.

Inherited-members

Methods

`fit`(dataset, *args, **kwargs)	Fit model.
`recommend`(users, dataset, k, filter_viewed)	Recommend items for users.
`recommend_to_items`(target_items, dataset, k)	Recommend items for target items.

Attributes

`recommends_for_cold`
`recommends_for_warm`

PureSVDModel

class rectools.models.pure_svd.**PureSVDModel**(*factors: int = 10*, *verbose: int = 0*)

Bases: *VectorModel*

PureSVD matrix factorization model.

See https://dl.acm.org/doi/10.1145/1864708.1864721

Parameters

- **factors** (int, default `10`) – The number of latent factors to compute.
- **verbose** (int, default `0`) – Degree of verbose output. If `0`, no output will be provided.

Inherited-members

Methods

`fit`(dataset, *args, **kwargs)	Fit model.
get_vectors()	Return user and item vector representations from fitted model.
`recommend`(users, dataset, k, filter_viewed)	Recommend items for users.
`recommend_to_items`(target_items, dataset, k)	Recommend items for target items.

Attributes

`i2i_dist`
`n_threads`
`recommends_for_cold`
`recommends_for_warm`
`u2i_dist`

`get_vectors()` → Tuple[ndarray, ndarray]

Return user and item vector representations from fitted model.

Returns
User and item embeddings. Shapes are (n_users, n_factors) and (n_items, n_factors).

Return type
(np.ndarray, np.ndarray)

RandomModel

`class rectools.models.random.RandomModel(random_state: Optional[int] = None, verbose: int = 0)`

Bases: *ModelBase*

Model generating random recommendations.

By default all items that are present in *dataset.item_id_map* will be used for recommendations.

Numbers ranging from <n recommendations for user> to 1 will be used as a "score" in recommendations.

Parameters

- **random_state** (int, optional, default `None`) – Pseudorandom number generator state to control the sampling.
- **verbose** (int, default `0`) – Degree of verbose output. If `0`, no output will be provided.

Inherited-members

Methods

`fit`(dataset, *args, **kwargs)	Fit model.
`recommend`(users, dataset, k, filter_viewed)	Recommend items for users.
`recommend_to_items`(target_items, dataset, k)	Recommend items for target items.

Attributes

`recommends_for_cold`
`recommends_for_warm`

What are you waiting for? Train and apply them!

Recommendation Table

Recommendation table contains recommendations for each user. It has a fixed set of columns, though they are different for i2i and u2i recommendations. Recommendation table can also be used for calculation of metrics.

3.4 Metrics

3.4.1 Details of RecTools Metrics

See the API documentation for further details on Dataset:

rectools.metrics.classification.Accuracy(k)	Ratio of correctly recommended items among all items.
rectools.metrics.popularity.AvgRecPopularity(k)	Average Recommendations Popularity metric.
rectools.metrics.classification.F1Beta(k[, beta])	Fbeta score for k first recommendations.
rectools.metrics.classification.HitRate(k)	HitRate calculates the fraction of users for which the correct answer is included in the recommendation list.
rectools.metrics.diversity.IntraListDiversity(k, ...)	Intra-List Diversity metric.
rectools.metrics.ranking.MAP(k[, divide_by_k])	Mean Average Precision at k (MAP@k).
rectools.metrics.classification.MCC(k)	Matthew correlation coefficient calculates correlation between actual and predicted classification.
rectools.metrics.ranking.MRR(k)	Mean Reciprocal Rank at k (MRR@k).
rectools.metrics.novelty.MeanInvUserFreq(k)	Mean Inverse User Frequency metric.
rectools.metrics.ranking.NDCG(k[, log_base])	Normalized Discounted Cumulative Gain at k (NDCG@k).
rectools.metrics.distances.PairwiseDistanceCalculator()	Base pairwise distance calculator class
rectools.metrics.distances.PairwiseHammingDistanceCalculator(...)	Class for computing Hamming distance between a pair of items.
rectools.metrics.classification.Precision(k)	Ratio of relevant items among top-k recommended items.
rectools.metrics.classification.Recall(k)	Ratio of relevant recommended items among all items user interacted with after recommendations were made.
rectools.metrics.serendipity.Serendipity(k)	Serendipity metric.
rectools.metrics.distances.SparsePairwiseHammingDistanceCalculator(...)	Class for computing Hamming distance between multiple pairs of elements represented in features matrix in sparse form.
rectools.metrics.scoring.calc_metrics(...[, ...])	Calculate metrics.

Accuracy

class rectools.metrics.classification.**Accuracy**(*k: int*)

Bases: *ClassificationMetric*

Ratio of correctly recommended items among all items.

The accuracy@k equals to `(tp + tn) / n_items` where

- `tp` is the number of relevant recommendations among the first `k` items in recommendation list;
- `tn` is the number of items with which user has not interacted (bought, liked) with (in period after recommendations were given) and we do not recommend to him (in the top `k` items of recommendation list);
- `n_items` - an overall number of items that could be used for recommendations.

Parameters

k (*int*) – Number of items at the top of recommendations list that will be used to calculate metric.

Inherited-members

Methods

calc(reco, interactions, catalog)	Calculate metric value.
calc_from_confusion_df(confusion_df, catalog)	Calculate metric value from prepared confusion matrix.
calc_per_user(reco, interactions, catalog)	Calculate metric values for all users.
calc_per_user_from_confusion_df(...)	Calculate metric values for all users from prepared confusion matrix.

Attributes

AvgRecPopularity

class rectools.metrics.popularity.**AvgRecPopularity**(*k: int*, *normalize: bool = False*)

Bases: *MetricAtK*

Average Recommendations Popularity metric.

Calculate the average popularity of the recommended items in each list, where "popularity" of an item is the number of previous interactions with this item.

$$ARP@k = \frac{1}{|U_t|} \sum_{u \in U_t} \frac{\sum_{i \in L_u} \phi(i)}{|L_u|}$$

$$NormalizedARP@k = \frac{1}{|U_t|} \sum_{u \in U_t} \frac{(\sum_{i \in L_u} \phi(i))/|interactions|}{|L_u|}$$

where

- $\phi(i)$ is the number of previous interactions with item i;

- $|U_t|$ is the number of users in the test set;
- $|interactions|$ is the total number of interactions;
- L_u is the list of top k recommended items for user u.

Parameters

- **k** (*int*) – Number of items at the top of recommendations list that will be used to calculate metric.
- **normalize** (*bool*) – Flag, which says whether to normalize metric or not. Normalization is done on total items popularity. This gives a probabilistic interpretation of the metric that can be easily applied to any data.

Examples

```
>>> reco = pd.DataFrame(
...     {
...         Columns.User: [1, 1, 2, 2, 2, 3, 3],
...         Columns.Item: [1, 2, 3, 1, 2, 3, 2],
...         Columns.Rank: [1, 2, 1, 2, 3, 1, 2],
...     }
... )
>>> prev_interactions = pd.DataFrame(
...     {
...         Columns.User: [1, 1, 2, 2, 3, 3],
...         Columns.Item: [1, 2, 1, 3, 1, 2],
...     }
... )
>>> AvgRecPopularity(k=1).calc_per_user(reco, prev_interactions).values
array([3., 1., 1.])
>>> AvgRecPopularity(k=3).calc_per_user(reco, prev_interactions).values
array([2.5, 2. , 1.5])
>>> AvgRecPopularity(k=3, normalize=True).calc_per_user(reco, prev_interactions).
↪values
array([0.41666667, 0.33333333, 0.25        ])
```

Inherited-members
Parameters

- **k** (*int*) –
- **normalize** (*bool*) –

Methods

calc(reco, prev_interactions)	Calculate metric value.
calc_per_user(reco, prev_interactions)	Calculate metric values for all users.

Attributes

normalize

calc(*reco: DataFrame*, *prev_interactions: DataFrame*) → float

Calculate metric value.

Parameters

- **reco** (*pd.DataFrame*) – Recommendations table with columns *Columns.User*, *Columns.Item*, *Columns.Rank*.
- **prev_interactions** (*pd.DataFrame*) – Table with previous user-item interactions, with columns *Columns.User*, *Columns.Item*.

Returns
Value of metric (average between users).

Return type
float

calc_per_user(*reco: DataFrame*, *prev_interactions: DataFrame*) → Series

Calculate metric values for all users.

Parameters

- **reco** (*pd.DataFrame*) – Recommendations table with columns *Columns.User*, *Columns.Item*, *Columns.Rank*.
- **prev_interactions** (*pd.DataFrame*) – Table with previous user-item interactions, with columns *Columns.User*, *Columns.Item*.

Returns
Values of metric (index - user id, values - metric value for every user).

Return type
pd.Series

F1Beta

class rectools.metrics.classification.**F1Beta**(*k: int*, *beta: float = 1.0*)

Bases: *SimpleClassificationMetric*

Fbeta score for k first recommendations. See more: https://en.wikipedia.org/wiki/F-score

The f1_beta equals to `(1 + beta_sqr) * p@k * r@k / (beta_sqr * p@k + r@k)` where

- beta_sqr equals to beta ** 2
- **p@k: precision@k equals to `tp / k` where**

-tp is the number of relevant recommendations
among first k items in the top of recommendation list.

- **r@k: recall@k equals to tp / liked where**
  - **tp is the number of relevant recommendations**
    among first k items in the top of recommendation list;
  - **liked is the number of items the user has interacted**
    (bought, liked) with (in period after recommendations were given).

Parameters

- **k** (*int*) – Number of items in top of recommendations list that will be used to calculate metric.
- **beta** (*float*) – Weight of recall. Default value: beta = 1.0

Inherited-members

Methods

calc(reco, interactions)	Calculate metric value.
calc_from_confusion_df(confusion_df)	Calculate metric value from prepared confusion matrix.
calc_per_user(reco, interactions)	Calculate metric values for all users.
calc_per_user_from_confusion_df(confusion_df)	Calculate metric values for all users from prepared confusion matrix.

Attributes

beta

HitRate

class rectools.metrics.classification.**HitRate**(*k: int*)

Bases: *SimpleClassificationMetric*

HitRate calculates the fraction of users for which the correct answer is included in the recommendation list.

The HitRate equals to 1 if tp > 0, otherwise 0 where

- tp is the number of relevant recommendations among the first k items in recommendation list.

Parameters
k (*int*) – Number of items in top of recommendations list that will be used to calculate metric.

Inherited-members

Methods

`calc`(reco, interactions)	Calculate metric value.
`calc_from_confusion_df`(confusion_df)	Calculate metric value from prepared confusion matrix.
`calc_per_user`(reco, interactions)	Calculate metric values for all users.
`calc_per_user_from_confusion_df`(confusion_df)	Calculate metric values for all users from prepared confusion matrix.

Attributes

IntraListDiversity

class rectools.metrics.diversity.**IntraListDiversity**(*k: int*, *distance_calculator:* PairwiseDistanceCalculator)

Bases: *MetricAtK*

Intra-List Diversity metric.

Estimate average pairwise distance between items in user recommendations.

$$ILD@k = (\sum_{i=1}^{k+1}\sum_{j=1}^{k+1} d(i,j))/(k*(k-1))$$

where - `d(i, j)` is distance between recommended items with rank `i` and rank `j`.

Parameters

- **k** (*int*) – Number of items at the top of recommendations list that will be used to calculate metric.
- **distance_calculator** (PairwiseDistanceCalculator) – Distance calculator, object that returns distance between any item pair.

Examples

```
>>> from rectools.metrics.distances import PairwiseHammingDistanceCalculator
>>> reco = pd.DataFrame(
...     {
...         Columns.User: [1, 1, 1, 2, 2],
...         Columns.Item: [1, 2, 3, 1, 4],
...         Columns.Rank: [1, 2, 3, 1, 2],
...     }
... )
>>> features_df = pd.DataFrame(
...     [
...         [1, 0, 0],
...         [2, 0, 1],
...         [3, 1, 1],
...         [4, 0, 0],
```

(continues on next page)

(continued from previous page)

```
...     ],
...     columns=[Columns.Item, "feature_1", "feature_2"]
... ).set_index(Columns.Item)
>>> calculator = PairwiseHammingDistanceCalculator(features_df)
>>> IntraListDiversity(k=1, distance_calculator=calculator).calc_per_user(reco).
↪values
array([0, 0])
>>> IntraListDiversity(k=2, distance_calculator=calculator).calc_per_user(reco).
↪values
array([1., 0.])
>>> IntraListDiversity(k=3, distance_calculator=calculator).calc_per_user(reco).
↪values
array([1.33333333, 0.        ])
```

Inherited-members

Parameters

- **k** (*int*) –
- **distance_calculator** (PairwiseDistanceCalculator) –

Methods

calc(reco)	Calculate metric value.
calc_from_fitted(fitted)	Calculate metric value from fitted data.
calc_per_user(reco)	Calculate metric values for all users.
calc_per_user_from_fitted(fitted)	Calculate metric values for all users from fitted data.
fit(reco, k_max)	Prepare intermediate data for effective calculation.

Attributes

distance_calculator

calc(*reco: DataFrame*) → float

Calculate metric value.

Parameters

reco (*pd.DataFrame*) – Recommendations table with columns *Columns.User*, *Columns.Item*, *Columns.Rank*.

Returns

Value of metric (average between users).

Return type

float

calc_from_fitted(*fitted:* ILDFitted) → float

Calculate metric value from fitted data.

For parameters used result of *fit* method.

Parameters
fitted (ILDFitted) – Meta data that got from *.fit* method.

Returns
Value of metric (average between users).

Return type
float

calc_per_user(*reco: DataFrame*) → Series

Calculate metric values for all users.

Parameters
reco (*pd.DataFrame*) – Recommendations table with columns *Columns.User*, *Columns.Item*, *Columns.Rank*.

Returns
Values of metric (index - user id, values - metric value for every user).

Return type
pd.Series

calc_per_user_from_fitted(*fitted:* ILDFitted) → Series

Calculate metric values for all users from fitted data.

For parameters used result of *fit* method.

Parameters
fitted (ILDFitted) – Meta data that got from *.fit* method.

Returns
Values of metric (index - user id, values - metric value for every user).

Return type
pd.Series

classmethod fit(*reco: DataFrame*, *k_max: int*) → *ILDFitted*

Prepare intermediate data for effective calculation.

You can use this method to prepare some intermediate data for later calculation. It can optimize calculations if you want calculate metric value for different *k* or *distance_calculator*.

Parameters

- **reco** (*pd.DataFrame*) – Recommendations table with columns *Columns.User*, *Columns.Item*, *Columns.Rank*.
- **k_max** (*int*) – k is number of items at the top of recommendations list that will be used to calculate metric. So *k_max* is maximum value of *k* metric for that you want to calculate.

Return type
ILDFitted

MAP

class rectools.metrics.ranking.**MAP**(*k: int*, *divide_by_k: bool = False*)

Bases: *_RankingMetric*

Mean Average Precision at k (MAP@k).

Mean AP calculates as mean value of AP among all users.

Average Precision estimates precision of recommendations taking into account their order.

$$AP@k = (\sum_{i=1}^{k+1} p@i * rel(i))/divider$$

where

- $p@i$ is `precision at i`, see *Precision* metric documentation for details;
- *rel(i)* is an indicator function, it equals to `1` if the item at rank `i` is relevant, `0` otherwise;
- *divider* can be equal to `k` or be equal to number of relevant items per user, depending on *divide_by_k* parameter.

Parameters

- **k** (*int*) – Number of items at the top of recommendations list that will be used to calculate metric.
- **divide_by_k** (*bool, default False*) – If `True`, `k` will be used as divider in `AP@k`. If `False`, number of relevant items for each user will be used.

Examples

```
>>> reco = pd.DataFrame(
...     {
...         Columns.User: [1, 1, 2, 2, 3, 3, 3, 3, 4, 4, 4],
...         Columns.Item: [7, 8, 1, 2, 1, 2, 3, 4, 1, 2, 3],
...         Columns.Rank: [1, 2, 1, 2, 1, 2, 3, 4, 1, 2, 3],
...     }
... )
>>> interactions = pd.DataFrame(
...     {
...         Columns.User: [1, 1, 2, 3, 3, 3, 4, 4, 4],
...         Columns.Item: [1, 2, 1, 1, 3, 4, 1, 2, 3],
...     }
... )
>>> # Here
>>> #    - for user ``1`` we return non-relevant recommendations;
>>> #    - for user ``2`` we return 2 items and relevant is first;
>>> #    - for user ``3`` we return 4 items, 1st, 3rd and 4th are relevant;
>>> #    - for user ``4`` we return 3 items and all are relevant;
>>> MAP(k=1).calc_per_user(reco, interactions).values
array([0. , 1. , 0.33333333, 0.33333333])
>>> MAP(k=3).calc_per_user(reco, interactions).values
array([0. , 1. , 0.55555556, 1. ])
>>> MAP(k=1, divide_by_k=True).calc_per_user(reco, interactions).values
```

(continues on next page)

(continued from previous page)

```
array([0., 1., 1., 1.])
>>> MAP(k=3, divide_by_k=True).calc_per_user(reco, interactions).values
array([0. , 0.33333333, 0.55555556, 1. ])
```

Inherited-members

Parameters

- **k** (*int*) –
- **divide_by_k** (*bool*) –

Methods

calc(reco, interactions)	Calculate metric value.
calc_from_fitted(fitted)	Calculate metric value from fitted data.
calc_per_user(reco, interactions)	Calculate metric values for all users.
calc_per_user_from_fitted(fitted)	Calculate metric values for all users from fitted data.
fit(merged, k_max)	Prepare intermediate data for effective calculation.

Attributes

divide_by_k

calc_from_fitted(*fitted:* MAPFitted) → float

Calculate metric value from fitted data.

For parameters used result of *fit* method.

Parameters

fitted (MAPFitted) – Meta data that got from *.fit* method.

Returns

Value of metric (average between users).

Return type

float

calc_per_user(*reco: DataFrame*, *interactions: DataFrame*) → Series

Calculate metric values for all users.

Parameters

- **reco** (*pd.DataFrame*) – Recommendations table with columns *Columns.User*, *Columns.Item*, *Columns.Rank*.
- **interactions** (*pd.DataFrame*) – Interactions table with columns *Columns.User*, *Columns.Item*.

Returns

Values of metric (index - user id, values - metric value for every user).

Return type

pd.Series

calc_per_user_from_fitted(*fitted:* MAPFitted) → Series

Calculate metric values for all users from fitted data.

For parameters used result of *fit* method.

Parameters
fitted (MAPFitted) – Meta data that got from *.fit* method.

Returns
Values of metric (index - user id, values - metric value for every user).

Return type
pd.Series

classmethod fit(*merged: DataFrame*, *k_max: int*) → *MAPFitted*

Prepare intermediate data for effective calculation.

You can use this method to prepare some intermediate data for later calculation. It can optimize calculations if you want calculate metric value for different *k*.

Parameters

- **merged** (*pd.DataFrame*) – Result of merging recommendations and interactions tables. Can be obtained using *merge_reco* function.
- **k_max** (*int*) – k is number of items at the top of recommendations list that will be used to calculate metric. So *k_max* is maximum number of items for which you want to calculate metric.

Return type
MAPFitted

MCC

class rectools.metrics.classification.**MCC**(*k: int*)

Bases: *ClassificationMetric*

Matthew correlation coefficient calculates correlation between actual and predicted classification. Min value = -1 (negative correlation), Max value = 1 (positive correlation), zero means no correlation See more: https://en.wikipedia.org/wiki/Phi_coefficient

The MCC equals to `(tp * tn - fp * fn) / sqrt((tp + fp)(tp + fn)(tn + fp)(tn + fn))` where

- `tp` is the number of relevant recommendations among the first `k` items in recommendation list;
- `tn` is the number of items with which user has not interacted (bought, liked) with (in period after recommendations were given) and we do not recommend to him (in the top `k` items of recommendation list);
- `fp` - number of non-relevant recommendations among the first *k* items of recommendation list;
- `fn` - number of items the user has interacted with but that weren't recommended (in top-*k*).

Parameters
k (*int*) – Number of items in top of recommendations list that will be used to calculate metric.

Inherited-members

Methods

`calc`(reco, interactions, catalog)	Calculate metric value.
`calc_from_confusion_df`(confusion_df, catalog)	Calculate metric value from prepared confusion matrix.
`calc_per_user`(reco, interactions, catalog)	Calculate metric values for all users.
`calc_per_user_from_confusion_df`(...)	Calculate metric values for all users from prepared confusion matrix.

Attributes

MRR

class rectools.metrics.ranking.**MRR**(*k: int*)

Bases: *_RankingMetric*

Mean Reciprocal Rank at k (MRR@k).

MRR calculates as mean value of reciprocal rank of first relevant recommendation among all users.

Estimates relevance of recommendations taking in account their order.

$$MRR@K = \frac{1}{|U|}\sum_{i=1}^{|U|}\frac{1}{rank_i}$$

where

- $|U|$ is a number of unique users;
- $rank_i$ is a rank of first relevant recommendation starting from `1`.

If a user doesn't have any relevant recommendation then his metric value will be `0`.

Parameters

k (*int*) – Number of items at the top of recommendations list that will be used to calculate metric.

Examples

```
>>> reco = pd.DataFrame(
...     {
...         Columns.User: [1, 1, 2, 2, 3, 3, 3, 3, 4, 4, 4],
...         Columns.Item: [7, 8, 1, 2, 2, 1, 3, 4, 7, 8, 3],
...         Columns.Rank: [1, 2, 1, 2, 1, 2, 3, 4, 1, 2, 3],
...     }
... )
>>> interactions = pd.DataFrame(
...     {
...         Columns.User: [1, 1, 2, 3, 3, 3, 4, 4, 4],
...         Columns.Item: [1, 2, 1, 1, 3, 4, 1, 2, 3],
...     }
```

(continues on next page)

(continued from previous page)

```
... )
>>> # Here
>>> #    - for user ``1`` we return non-relevant recommendations;
>>> #    - for user ``2`` we return 2 items and relevant is first;
>>> #    - for user ``3`` we return 4 items, 2nd, 3rd and 4th are relevant;
>>> #    - for user ``4`` we return 3 items and relevant is last;
>>> MRR(k=1).calc_per_user(reco, interactions).values
array([0., 1., 0., 0.])
>>> MRR(k=3).calc_per_user(reco, interactions).values
array([0.        , 1.        , 0.5       , 0.33333333])
```

Inherited-members

Parameters

k (*int*) –

Methods

calc(reco, interactions)	Calculate metric value.
calc_from_merged(merged)	Calculate metric value from merged recommendations.
calc_per_user(reco, interactions)	Calculate metric values for all users.
calc_per_user_from_merged(merged)	Calculate metric values for all users from merged recommendations.

Attributes

calc_from_merged(*merged: DataFrame*) → float

Calculate metric value from merged recommendations.

Parameters

merged (*pd.DataFrame*) – Result of merging recommendations and interactions tables. Can be obtained using *merge_reco* function.

Returns

Value of metric (average between users).

Return type

float

calc_per_user(*reco: DataFrame*, *interactions: DataFrame*) → Series

Calculate metric values for all users.

Parameters

- **reco** (*pd.DataFrame*) – Recommendations table with columns *Columns.User*, *Columns.Item*, *Columns.Rank*.
- **interactions** (*pd.DataFrame*) – Interactions table with columns *Columns.User*, *Columns.Item*.

Returns
Values of metric (index - user id, values - metric value for every user).

Return type
pd.Series

`calc_per_user_from_merged`(*merged: DataFrame*) → Series

Calculate metric values for all users from merged recommendations.

Parameters
merged (*pd.DataFrame*) – Result of merging recommendations and interactions tables. Can be obtained using *merge_reco* function.

Returns
Values of metric (index - user id, values - metric value for every user).

Return type
pd.Series

MeanInvUserFreq

`class rectools.metrics.novelty.MeanInvUserFreq`(*k: int*)

Bases: *MetricAtK*

Mean Inverse User Frequency metric.

Estimate mean novelty of items in recommendations, where "novelty" of item is inversely proportional to the number of users who interacted with it.

$$MIUF@k = -(\sum_{i=1}^{k} \log_2(users(i)/n\_users))/k$$

where - *users(i)* is number of users that previously interacted with item with rank *i*. - *n_users* is the overall number of users in previous interactions.

Parameters
k (*int*) – Number of items at the top of recommendations list that will be used to calculate metric.

Examples

```
>>> reco = pd.DataFrame(
...     {
...         Columns.User: [1, 2, 2, 3, 3],
...         Columns.Item: [3, 2, 3, 1, 2],
...         Columns.Rank: [1, 1, 2, 1, 2],
...     }
... )
>>> prev_interactions = pd.DataFrame(
...     {
...         Columns.User: [1, 1, 2, 3],
...         Columns.Item: [1, 2, 1, 1],
...     }
... )
>>> MeanInvUserFreq(k=1).calc_per_user(reco, prev_interactions).values
```

(continues on next page)

(continued from previous page)

```
array([1.5849625, 1.5849625, 0. ])
>>> MeanInvUserFreq(k=3).calc_per_user(reco, prev_interactions).values
array([1.5849625 , 1.5849625 , 0.79248125])
```

Inherited-members

Parameters

k (*int*) –

Methods

calc(reco, prev_interactions)	Calculate metric value.
calc_from_fitted(fitted)	Calculate metric value from fitted data.
calc_per_user(reco, prev_interactions)	Calculate metric values for all users.
calc_per_user_from_fitted(fitted)	Calculate metric values for all users from fitted data.
fit(reco, prev_interactions, k_max)	Prepare intermediate data for effective calculation.

Attributes

calc(*reco: DataFrame*, *prev_interactions: DataFrame*) → float

Calculate metric value.

Parameters

- **reco** (*pd.DataFrame*) – Recommendations table with columns *Columns.User*, *Columns.Item*, *Columns.Rank*.
- **prev_interactions** (*pd.DataFrame*) – Table with previous user-item interactions, with columns *Columns.User*, *Columns.Item*.

Returns

Value of metric (average between users).

Return type

float

calc_from_fitted(*fitted:* MIUFFitted) → float

Calculate metric value from fitted data.

For parameters used result of *fit* method.

Parameters

fitted (MIUFFitted) – Meta data that got from *.fit* method.

Returns

Value of metric (average between users).

Return type

float

calc_per_user(*reco: DataFrame*, *prev_interactions: DataFrame*) → Series

Calculate metric values for all users.

Parameters

- **reco** (*pd.DataFrame*) – Recommendations table with columns *Columns.User*, *Columns.Item*, *Columns.Rank*.
- **prev_interactions** (*pd.DataFrame*) – Table with previous user-item interactions, with columns *Columns.User*, *Columns.Item*.

Returns
Values of metric (index - user id, values - metric value for every user).

Return type
pd.Series

calc_per_user_from_fitted(*fitted:* MIUFFitted) → Series

Calculate metric values for all users from fitted data.

For parameters used result of *fit* method.

Parameters
fitted (MIUFFitted) – Meta data that got from *.fit* method.

Returns
Values of metric (index - user id, values - metric value for every user).

Return type
pd.Series

classmethod fit(*reco: DataFrame*, *prev_interactions: DataFrame*, *k_max: int*) → *MIUFFitted*

Prepare intermediate data for effective calculation.

You can use this method to prepare some intermediate data for later calculation. It can optimize calculations if you want calculate metric for different values of *k*.

Parameters

- **reco** (*pd.DataFrame*) – Recommendations table with columns *Columns.User*, *Columns.Item*, *Columns.Rank*.
- **prev_interactions** (*pd.DataFrame*) – Table with previous user-item interactions, with columns *Columns.User*, *Columns.Item*.
- **k_max** (*int*) – *k* is number of items at the top of recommendations list that will be used to calculate metric. So *k_max* is maximum value of *k* parameter for which you want to calculate metric.

Return type
MIUFFitted

NDCG

class rectools.metrics.ranking.**NDCG**(*k: int*, *log_base: int = 2*)

Bases: *_RankingMetric*

Normalized Discounted Cumulative Gain at k (NDCG@k).

Estimates relevance of recommendations taking in account their order.

$$NDCG@k = DCG@k / IDCG@k$$

where $DCG@k = \sum_{i=1}^{k+1} rel(i)/log(i+1)$ - Discounted Cumulative Gain at k, main part of *NDCG@k*.

The closer it is to the top the more weight it assigns to relevant items. Here: - *rel(i)* is an indicator function, it equals to 1 if an item at rank *i* is relevant, 0 otherwise; - *log* - logarithm at any given base, usually 2.

and $IDCG@k = \sum_{i=1}^{k+1}(1/log(i+1))$ - *Ideal DCG@k*, maximum possible value of *DCG@k*, used as normalization coefficient to ensure that *NDCG@k* values lie in `[0, 1]`.

Parameters

- **k** (*int*) – Number of items at the top of recommendations list that will be used to calculate metric.
- **log_base** (int, default 2) – Base of logarithm used to weight relevant items.

Examples

```
>>> reco = pd.DataFrame(
...     {
...         Columns.User: [1, 1, 2, 2, 3, 3, 3, 3, 4, 4, 4],
...         Columns.Item: [7, 8, 1, 2, 1, 2, 3, 4, 1, 2, 3],
...         Columns.Rank: [1, 2, 1, 2, 1, 2, 3, 4, 1, 2, 3],
...     }
... )
>>> interactions = pd.DataFrame(
...     {
...         Columns.User: [1, 1, 2, 3, 3, 3, 4, 4, 4],
...         Columns.Item: [1, 2, 1, 1, 3, 4, 1, 2, 3],
...     }
... )
>>> # Here
>>> #    - for user ``1`` we return non-relevant recommendations;
>>> #    - for user ``2`` we return 2 items and relevant is first;
>>> #    - for user ``3`` we return 4 items, 1st, 3rd and 4th are relevant;
>>> #    - for user ``4`` we return 3 items and all are relevant;
>>> NDCG(k=1).calc_per_user(reco, interactions).values
array([0., 1., 1., 1.])
>>> NDCG(k=3).calc_per_user(reco, interactions).values
array([0.  , 0.46927873, 0.70391809, 1.  ])
```

Inherited-members
Parameters

- **k** (*int*) –
- **log_base** (*int*) –

Methods

calc(reco, interactions)	Calculate metric value.
calc_from_merged(merged)	Calculate metric value from merged recommendations.
calc_per_user(reco, interactions)	Calculate metric values for all users.
calc_per_user_from_merged(merged)	Calculate metric values for all users from merged recommendations.

Attributes

log_base

calc_from_merged(*merged: DataFrame*) → float

Calculate metric value from merged recommendations.

Parameters

merged (*pd.DataFrame*) – Result of merging recommendations and interactions tables. Can be obtained using *merge_reco* function.

Returns

Value of metric (average between users).

Return type

float

calc_per_user(*reco: DataFrame*, *interactions: DataFrame*) → Series

Calculate metric values for all users.

Parameters

- **reco** (*pd.DataFrame*) – Recommendations table with columns *Columns.User*, *Columns.Item*, *Columns.Rank*.
- **interactions** (*pd.DataFrame*) – Interactions table with columns *Columns.User*, *Columns.Item*.

Returns

Values of metric (index - user id, values - metric value for every user).

Return type

pd.Series

calc_per_user_from_merged(*merged: DataFrame*) → Series

Calculate metric values for all users from merged recommendations.

Parameters

merged (*pd.DataFrame*) – Result of merging recommendations and interactions tables. Can be obtained using *merge_reco* function.

Returns

Values of metric (index - user id, values - metric value for every user).

Return type

pd.Series

PairwiseDistanceCalculator

class rectools.metrics.distances.**PairwiseDistanceCalculator**

Bases: ABC

Base pairwise distance calculator class

Inherited-members

PairwiseHammingDistanceCalculator

class rectools.metrics.distances.**PairwiseHammingDistanceCalculator**(*item_features_df: DataFrame*)

Bases: *PairwiseDistanceCalculator*

Class for computing Hamming distance between a pair of items.

Parameters

item_features_df (*pandas dataframe with feature values and item ids as index*) –

Inherited-members

Precision

class rectools.metrics.classification.**Precision**(*k: int*)

Bases: *SimpleClassificationMetric*

Ratio of relevant items among top-*k* recommended items.

The precision@k equals to `tp / k` where `tp` is the number of relevant recommendations among first `k` items in the top of recommendation list.

Parameters

k (*int*) – Number of items in top of recommendations list that will be used to calculate metric.

Inherited-members

Methods

calc(reco, interactions)	Calculate metric value.
calc_from_confusion_df(confusion_df)	Calculate metric value from prepared confusion matrix.
calc_per_user(reco, interactions)	Calculate metric values for all users.
calc_per_user_from_confusion_df(confusion_df)	Calculate metric values for all users from prepared confusion matrix.

Attributes

Recall

class rectools.metrics.classification.**Recall**(*k: int*)

Bases: *SimpleClassificationMetric*

Ratio of relevant recommended items among all items user interacted with after recommendations were made.

The recall@k equals to `tp / liked` where

- `tp` is the number of relevant recommendations among first `k` items in the top of recommendation list;
- `liked` is the number of items the user has interacted (bought, liked) with (in period after recommendations were given).

Parameters

k (*int*) – Number of items in top of recommendations list that will be used to calculate metric.

Inherited-members

Methods

`calc`(reco, interactions)	Calculate metric value.
`calc_from_confusion_df`(confusion_df)	Calculate metric value from prepared confusion matrix.
`calc_per_user`(reco, interactions)	Calculate metric values for all users.
`calc_per_user_from_confusion_df`(confusion_df)	Calculate metric values for all users from prepared confusion matrix.

Attributes

Serendipity

class rectools.metrics.serendipity.**Serendipity**(*k: int*)

Bases: *MetricAtK*

Serendipity metric.

Evaluates novelty and relevance together.

$$Serendipity@k = (\sum_{i=1}^{k} max(p(i) - pu(i), 0) * rel(i))/k$$

where

- $p(i) = (n\_items + 1 - i)/n\_items$ is probability to recommend item with rank `i` to current user;
- $pu(i) = (n\_items + 1 - popularity(i))/n_items$ is probability to recommend item with rank `i` to any user;
- $rel(i)$ is an indicator function, it equals to `1` if the item at rank `i` is relevant, `0` otherwise;
- n_items is an overall number of items that could be used for recommendations.
- $popularity(i)$ is popularity rank of the i-th item in recommendations list.

Parameters

k (*int*) – Number of items at the top of recommendations list that will be used to calculate metric.

Notes

Method is inspired by the article: https://gab41.lab41.org/recommender-systems-its-not-all-about-the-accuracy-562c7dceeaff

Examples

```
>>> reco = pd.DataFrame(
...     {
...         Columns.User: ["u1", "u1", "u2", "u2", "u3", "u4", "u4"],
...         Columns.Item: ["i1", "i2", "i2", "i3", "i3", "i2", "i3"],
...         Columns.Rank: [   1,    2,    1,    2,    1,    1,    2],
...     }
... )
>>> interactions = pd.DataFrame(
...     {
...         Columns.User: ["u1", "u1", "u2", "u2", "u3", "u4"],
...         Columns.Item: ["i1", "i2", "i2", "i3", "i2", "i2"],
...     }
... )
>>> prev_interactions = pd.DataFrame(
...     {
...         Columns.User: ["u1", "u1", "u2", "u2", "u3"],
...         Columns.Item: ["i1", "i2", "i1", "i2", "i1"],
...     }
... )
>>> catalog = ("i1", "i2", "i3", "i4")
>>> Serendipity(k=1).calc_per_user(reco, interactions, prev_interactions, catalog).
↪values
array([0. , 0.25, 0. , 0.25])
>>> Serendipity(k=2).calc_per_user(reco, interactions, prev_interactions, catalog).
↪values
array([0.  , 0.5 , 0.  , 0.125])
```

Inherited-members

Parameters

k (*int*) –

Methods

calc(reco, interactions, prev_interactions, ...)	Calculate metric value.
calc_from_fitted(fitted)	Calculate metric value from fitted data.
calc_per_user(reco, interactions, ...)	Calculate metric values for all users.
calc_per_user_from_fitted(fitted)	Calculate metric values for all users from fitted data.
fit(reco, interactions, prev_interactions, ...)	Prepare intermediate data for effective calculation.

Attributes

calc(*reco: DataFrame*, *interactions: DataFrame*, *prev_interactions: DataFrame*, *catalog: Collection[Union[str, int]]*) → float

Calculate metric value.

Parameters

- **reco** (*pd.DataFrame*) – Recommendations table with columns *Columns.User*, *Columns.Item*, *Columns.Rank*.
- **interactions** (*pd.DataFrame*) – Interactions table with columns *Columns.User*, *Columns.Item*.
- **prev_interactions** (*pd.DataFrame*) – Table with previous user-item interactions, with columns *Columns.User*, *Columns.Item*.
- **catalog** (*collection*) – Collection of unique item ids that could be used for recommendations.

Returns
Value of metric (average between users).

Return type
float

calc_from_fitted(*fitted:* SerendipityFitted) → float

Calculate metric value from fitted data.

For parameters used result of *fit* method.

Parameters
fitted (SerendipityFitted) – Meta data that got from *.fit* method.

Returns
Value of metric (average between users).

Return type
float

calc_per_user(*reco: DataFrame*, *interactions: DataFrame*, *prev_interactions: DataFrame*, *catalog: Collection[Union[str, int]]*) → Series

Calculate metric values for all users.

Parameters

- **reco** (*pd.DataFrame*) – Recommendations table with columns *Columns.User*, *Columns.Item*, *Columns.Rank*.
- **interactions** (*pd.DataFrame*) – Interactions table with columns *Columns.User*, *Columns.Item*.
- **prev_interactions** (*pd.DataFrame*) – Table with previous user-item interactions, with columns *Columns.User*, *Columns.Item*.
- **catalog** (*collection*) – Collection of unique item ids that could be used for recommendations.

Returns
Values of metric (index - user id, values - metric value for every user).

Return type
pd.Series

calc_per_user_from_fitted(*fitted:* SerendipityFitted) → Series

Calculate metric values for all users from fitted data.

For parameters used result of *fit* method.

Parameters
fitted (SerendipityFitted) – Meta data that got from *.fit* method.

Returns
Values of metric (index - user id, values - metric value for every user).

Return type
pd.Series

classmethod fit(*reco: DataFrame*, *interactions: DataFrame*, *prev_interactions: DataFrame*, *catalog: Collection[Union[str, int]]*, *k_max: int*) → *SerendipityFitted*

Prepare intermediate data for effective calculation.

You can use this method to prepare some intermediate data for later calculation. It can optimize calculations if you want calculate metric value for different *k* or *distance_calculator*.

Parameters

- **reco** (*pd.DataFrame*) – Recommendations table with columns *Columns.User*, *Columns.Item*, *Columns.Rank*.
- **interactions** (*pd.DataFrame*) – Interactions table with columns *Columns.User*, *Columns.Item*.
- **prev_interactions** (*pd.DataFrame*) – Table with previous user-item interactions, with columns *Columns.User*, *Columns.Item*.
- **catalog** (*collection*) – Collection of unique item ids that could be used for recommendations.
- **k_max** (*int*) – k is number of items at the top of recommendations list that will be used to calculate metric. So *k_max* is maximum value of *k* parameter for which you want to calculate metric.

Return type
SerendipityFitted

SparsePairwiseHammingDistanceCalculator

class rectools.metrics.distances.**SparsePairwiseHammingDistanceCalculator**(*features:* SparseFeatures, *id_map:* IdMap)

Bases: *PairwiseDistanceCalculator*

Class for computing Hamming distance between multiple pairs of elements represented in features matrix in sparse form.

ATTENTION! An incorrect value may occur for float type matrix because nonsafe comparison isung (!=)

Parameters

- **features** (SparseFeatures) – Storage for sparse features with csr_matrix as field where the row index is associated with the identifier by id_map.

- **id_map** (IdMap) – Mapper which include mapping info between external and internal representations for all identificators for which you're planning search distances.

Examples

```
>>> from scipy.sparse import csr_matrix
>>> from rectools.dataset import IdMap, SparseFeatures
>>> from rectools.metrics import SparsePairwiseHammingDistanceCalculator
>>> features_matrix = csr_matrix(
...     [
...         [0, 0],
...         [0, 1],
...         [1, 1],
...     ])
>>> features = SparseFeatures(values=features_matrix, names=["feature_1", "feature_2
↪"])
>>> mapper = IdMap.from_values(["i1", "i2", "i3", "i4", "i5"])
>>> calculator = SparsePairwiseHammingDistanceCalculator(features, mapper)
>>> calculator[
...    ["i1", "i1", "i1"],
...    ["i1", "i2", "i3"]
... ]
array([0., 1., 2.], dtype=float32)
```

Inherited-members
Parameters

- **features** (SparseFeatures) –
- **id_map** (IdMap) –

calc_metrics

rectools.metrics.scoring.**calc_metrics**(*metrics: Dict[str, MetricAtK]*, *reco: DataFrame*, *interactions: Optional[DataFrame] = None*, *prev_interactions: Optional[DataFrame] = None*, *catalog: Optional[Collection[Union[str, int]]] = None*) → Dict[str, float]

Calculate metrics.

Parameters

- **metrics** (*dict(str -> Metric)*) – Dict of metric objects to calculate, where key is metric name and value is metric object.
- **reco** (*pd.DataFrame*) – Recommendations table with columns *Columns.User*, *Columns.Item*, *Columns.Rank*.
- **interactions** (*pd.DataFrame, optional*) – Interactions table with columns *Columns.User*, *Columns.Item*. Obligatory only for some types of metrics.
- **prev_interactions** (*pd.DataFrame*) – Table with previous user-item interactions, with columns *Columns.User*, *Columns.Item*. Obligatory only for some types of metrics.
- **catalog** (*collection, optional*) – Collection of unique item ids that could be used for recommendations. Obligatory only if *ClassificationMetric* or *SerendipityMetric* instances present in *metrics*.

Returns
Dictionary where keys are the same with keys in *metrics* and values are metric calculation results.

Return type
dict(str->float)

Raises
ValueError – If obligatory argument for some metric not set.

Examples

```
>>> from rectools import Columns
>>> from rectools.metrics import Accuracy, NDCG
>>> reco = pd.DataFrame(
...     {
...         Columns.User: [1, 1, 2, 2, 3, 3, 3, 3, 4, 4, 4],
...         Columns.Item: [7, 8, 1, 2, 1, 2, 3, 4, 1, 2, 3],
...         Columns.Rank: [1, 2, 1, 2, 1, 2, 3, 4, 1, 2, 3],
...     }
... )
>>> interactions = pd.DataFrame(
...     {
...         Columns.User: [1, 1, 2, 3, 3, 3, 4, 4, 4],
...         Columns.Item: [1, 2, 1, 1, 3, 4, 1, 2, 3],
...         Columns.Datetime: [1, 1, 1, 1, 1, 2, 2, 2, 2],
...     }
... )
>>> split_dt = 2
>>> df_train = interactions.loc[interactions[Columns.Datetime] < split_dt]
>>> df_test = interactions.loc[interactions[Columns.Datetime] >= split_dt]
>>> metrics = {
...     'ndcg@1': NDCG(k=1),
...     'accuracy@1': Accuracy(k=1)
... }
>>> calc_metrics(
...     metrics,
...     reco=reco,
...     interactions=df_test,
...     prev_interactions=df_train,
...     catalog=df_train[Columns.Item].unique()
... )
{'accuracy@1': 0.3333333333333333, 'ndcg@1': 0.5}
```

Oops, yeah, can't forget about them.

3.5 Model selection

3.5.1 Details of RecTools Model selection

See the API documentation for further details on Model selection:

rectools.model_selection.last_n_split.LastNSplitter(n)	Splitter for cross-validation by leave-one-out / leave-k-out scheme (recent activity).
rectools.model_selection.random_split.RandomSplitter(...)	Slitter for cross-validation by random.
rectools.model_selection.splitter.Splitter([...])	Base class to construct data splitters.
rectools.model_selection.time_split.TimeRangeSplitter(...)	Splitter for cross-validation by leave-time-out scheme.
rectools.model_selection.cross_validate.cross_validate(...)	Run cross validation on multiple models with multiple metrics.

LastNSplitter

class rectools.model_selection.last_n_split.**LastNSplitter**(*n: int*, *n_splits: int = 1*, *filter_cold_users: bool = True*, *filter_cold_items: bool = True*, *filter_already_seen: bool = True*)

Bases: *Splitter*

Splitter for cross-validation by leave-one-out / leave-k-out scheme (recent activity). Generate train and test putting last n interactions for each user in test and all of his previous interactions in train. Cross-validation is achieved with sliding window over each users interactions history.

This technique may be used for sequential recommendation scenarios. It is common in research papers on sequential recommendations. But it doesn't fully prevent data leak from the future.

It is also possible to exclude cold users and items and already seen items.

Parameters

- **n** (*int*) – Number of interactions for each user that will be included in test.
- **n_splits** (*int, default 1*) – Number of test folds.
- **filter_cold_users** (bool, default True) – If *True*, users that are not present in train will be excluded from test. WARNING: both cold and warm users will be excluded from test.
- **filter_cold_items** (bool, default True) – If *True*, items that are not present in train will be excluded from test. WARNING: both cold and warm items will be excluded from test.
- **filter_already_seen** (bool, default True) – If True, pairs (user, item) that are present in train will be excluded from test.

Examples

```
>>> from rectools import Columns
>>> df = pd.DataFrame(
...     [
...         [1, 1, 1, "2021-09-01"], # 0
...         [1, 2, 1, "2021-09-02"], # 1
...         [1, 1, 1, "2021-09-03"], # 2
...         [1, 2, 1, "2021-09-04"], # 3
...         [1, 2, 1, "2021-09-05"], # 4
...         [2, 1, 1, "2021-08-20"], # 5
...         [2, 2, 1, "2021-08-21"], # 6
...         [2, 2, 1, "2021-08-22"], # 7
...     ],
...     columns=[Columns.User, Columns.Item, Columns.Weight, Columns.Datetime],
... ).astype({Columns.Datetime: "datetime64[ns]"})
>>> interactions = Interactions(df)
>>>
>>> splitter = LastNSplitter(2, 2, False, False, False)
>>> for train_ids, test_ids, _ in splitter.split(interactions):
...     print(train_ids, test_ids)
[0] [1 2 5]
[0 1 2 5] [3 4 6 7]
>>>
>>> splitter = LastNSplitter(2, 2, True, False, False)
>>> for train_ids, test_ids, _ in splitter.split(interactions):
...     print(train_ids, test_ids)
[0] [1 2]
[0 1 2 5] [3 4 6 7]
```

Inherited-members

Parameters

- **n** (*int*) –
- **n_splits** (*int*) –
- **filter_cold_users** (*bool*) –
- **filter_cold_items** (*bool*) –
- **filter_already_seen** (*bool*) –

Methods

`filter`(interactions, collect_fold_stats, ...)	Filter train and test indexes from one fold based on *filter_cold_users*, *filter_cold_items*,`filter_already_seen` class fields.
`split`(interactions[, collect_fold_stats])	Split interactions into folds and apply filtration to the result.

RandomSplitter

class rectools.model_selection.random_split.**RandomSplitter**(*test_fold_frac: float*, *n_splits: int = 1*, *random_state: Optional[int] = None*, *filter_cold_users: bool = True*, *filter_cold_items: bool = True*, *filter_already_seen: bool = True*)

Bases: *Splitter*

Slitter for cross-validation by random. Generate train and test folds with fixed test part ratio without intersections between test folds. Random splitting is applied to interactions. Users and items are not taken into account while preparing splits.

It is also possible to exclude cold users and items and already seen items.

Parameters

- **test_fold_frac** (*float*) – Relative size of test part, must be between 0. and 1.
- **n_splits** (*int, default 1*) – Number of test folds.
- **random_state** (*int, default None,*) – Controls randomness of each fold. Pass an int to get reproducible result across multiple *split* calls.
- **filter_cold_users** (bool, default `True`) – If *True*, users that are not present in train will be excluded from test. WARNING: both cold and warm users will be excluded from test.
- **filter_cold_items** (bool, default `True`) – If *True*, items that are not present in train will be excluded from test. WARNING: both cold and warm items will be excluded from test.
- **filter_already_seen** (bool, default `True`) – If *True*, pairs (user, item) that are present in train will be excluded from test.

Examples

```
>>> from rectools import Columns
>>> df = pd.DataFrame(
...     [
...         [1, 2, 1, "2021-09-01"],  # 0
...         [2, 1, 1, "2021-09-02"],  # 1
...         [2, 3, 1, "2021-09-03"],  # 2
...         [3, 2, 1, "2021-09-03"],  # 3
...         [3, 3, 1, "2021-09-04"],  # 4
...         [3, 4, 1, "2021-09-04"],  # 5
...         [1, 2, 1, "2021-09-05"],  # 6
...         [4, 2, 1, "2021-09-05"],  # 7
...     ],
...     columns=[Columns.User, Columns.Item, Columns.Weight, Columns.Datetime],
... ).astype({Columns.Datetime: "datetime64[ns]"})
>>> interactions = Interactions(df)
>>>
>>> splitter = RandomSplitter(test_fold_frac=0.25, random_state=42, n_splits=2,␣
↪filter_cold_users=False,
...                        filter_cold_items=False, filter_already_seen=False)
```

(continues on next page)

(continued from previous page)

```
>>> for train_ids, test_ids, _ in splitter.split(interactions):
...     print(train_ids, test_ids)
[2 7 6 1 5 0] [3 4]
[3 4 6 1 5 0] [2 7]
>>>
>>> splitter = RandomSplitter(test_fold_frac=0.25, random_state=42, n_splits=2,␣
↪filter_cold_users=True,
...                        filter_cold_items=True, filter_already_seen=True)
>>> for train_ids, test_ids, _ in splitter.split(interactions):
...     print(train_ids, test_ids)
[2 7 6 1 5 0] [3 4]
[3 4 6 1 5 0] [2]
```

Inherited-members
Parameters

- **test_fold_frac** (*float*) –
- **n_splits** (*int*) –
- **random_state** (*Optional[int]*) –
- **filter_cold_users** (*bool*) –
- **filter_cold_items** (*bool*) –
- **filter_already_seen** (*bool*) –

Methods

filter(interactions, collect_fold_stats, ...)	Filter train and test indexes from one fold based on *filter_cold_users*, *filter_cold_items*,`filter_already_seen` class fields.
split(interactions[, collect_fold_stats])	Split interactions into folds and apply filtration to the result.

Splitter

class rectools.model_selection.splitter.**Splitter**(*filter_cold_users: bool = True*, *filter_cold_items: bool = True*, *filter_already_seen: bool = True*)

Bases: object

Base class to construct data splitters. It cannot be used directly. New splitter can be defined by subclassing the *Splitter* class and implementing *_split_without_filter* method. Check specific class descriptions to get more information.

Inherited-members
Parameters

- **filter_cold_users** (*bool*) –
- **filter_cold_items** (*bool*) –
- **filter_already_seen** (*bool*) –

Methods

filter(interactions, collect_fold_stats, ...)	Filter train and test indexes from one fold based on *filter_cold_users*, *filter_cold_items*,`filter_already_seen` class fields.
split(interactions[, collect_fold_stats])	Split interactions into folds and apply filtration to the result.

filter(*interactions:* Interactions, *collect_fold_stats: bool*, *train_idx: ndarray*, *test_idx: ndarray*, *split_info: Dict[str, Any]*) → Tuple[ndarray, ndarray, Dict[str, Any]]

Filter train and test indexes from one fold based on *filter_cold_users*, *filter_cold_items*,`filter_already_seen` class fields. They are set to *True* by default.

Parameters

- **interactions** (*Interactions*) – User-item interactions.
- **collect_fold_stats** (*bool, default False*) – Add some stats to split info, like size of train and test part, number of users and items.
- **train_idx** (*array*) – Train part row numbers.
- **test_idx** (*array*) – Test part row numbers.
- **split_info** (*dict*) – Information about the split.

Returns

Returns tuple with filtered train part row numbers, test part row numbers and split info.

Return type

Tuple(array, array, dict)

split(*interactions:* Interactions, *collect_fold_stats: bool = False*) → Iterator[Tuple[ndarray, ndarray, Dict[str, Any]]]

Split interactions into folds and apply filtration to the result.

Parameters

- **interactions** (*Interactions*) – User-item interactions.
- **collect_fold_stats** (*bool, default False*) – Add some stats to split info, like size of train and test part, number of users and items.

Returns

Yields tuples with train part row numbers, test part row numbers and split info.

Return type

iterator(array, array, dict)

TimeRangeSplitter

class rectools.model_selection.time_split.**TimeRangeSplitter**(*test_size: str*, *n_splits: int = 1*, *filter_cold_users: bool = True*, *filter_cold_items: bool = True*, *filter_already_seen: bool = True*)

Bases: *Splitter*

Splitter for cross-validation by leave-time-out scheme. Generate train and test putting all interactions for all users after fixed date-time in test and all interactions before this date-time in train. Cross-validation is achieved with sliding window over timeline of interactions.

Size of the window is defined in days or hours. Test folds do not intersect and start one right after the other. This technique fully reproduces the real life scenario for recommender systems, preventing any data leak from the future.

It is advised to remember daily and weekly patterns in time series, making each fold equal to full day or full week when such patterns are present in data.

It is also possible to exclude cold users and items and already seen items.

Parameters

- **test_size** (*str*) – Size of test fold in format `[1-9]\d*[DH]`, e.g. `1D` (1 day), `4H` (4 hours). Test folds are taken from the end of *interactions*. The last fold includes the whole time unit with the last interaction. E.g. if the last interaction was at 01:25 a.m. of Monday, then with *test_size* = *"1D"* the last fold will be the full Monday, and with *test_size* = *"1H"* the last fold will be between 01:00 a.m. and 02:00 a.m on Monday.
- **n_splits** (*int*) – Number of test folds.
- **filter_cold_users** (bool, default `True`) – If *True*, users that are not present in train will be excluded from test. WARNING: both cold and warm users will be excluded from test.
- **filter_cold_items** (bool, default `True`) – If *True*, items that are not present in train will be excluded from test. WARNING: both cold and warm items will be excluded from test.
- **filter_already_seen** (bool, default `True`) – If `True`, pairs (user, item) that are present in train will be excluded from test.

Examples

```
>>> from datetime import date
>>> df = pd.DataFrame(
...     [
...         [1, 2, 1, "2021-09-01"],  # 0
...         [2, 1, 1, "2021-09-02"],  # 1
...         [2, 3, 1, "2021-09-03"],  # 2
...         [3, 2, 1, "2021-09-03"],  # 3
...         [3, 3, 1, "2021-09-04"],  # 4
...         [4, 4, 1, "2021-09-04"],  # 5
...         [1, 2, 1, "2021-09-05"],  # 6
...     ],
...     columns=[Columns.User, Columns.Item, Columns.Weight, Columns.Datetime],
... ).astype({Columns.Datetime: "datetime64[ns]"})
```

(continues on next page)

(continued from previous page)

```
>>> interactions = Interactions(df)
>>>
>>> splitter = TimeRangeSplitter("1D", 2, False, False, False)
>>> for train_ids, test_ids, _ in splitter.split(interactions):
...     print(train_ids, test_ids)
[0 1 2 3] [4 5]
[0 1 2 3 4 5] [6]
>>>
>>> splitter = TimeRangeSplitter("1D", 2, True, False, False)
>>> for train_ids, test_ids, _ in splitter.split(interactions):
...     print(train_ids, test_ids)
[0 1 2 3] [4]
[0 1 2 3 4 5] [6]
```

Inherited-members

Parameters

- **test_size** (*str*) –
- **n_splits** (*int*) –
- **filter_cold_users** (*bool*) –
- **filter_cold_items** (*bool*) –
- **filter_already_seen** (*bool*) –

Methods

filter(interactions, collect_fold_stats, ...)	Filter train and test indexes from one fold based on *filter_cold_users*, *filter_cold_items*,`filter_already_seen` class fields.
get_test_fold_borders(interactions)	Return datetime borders of test folds based on given test fold sizes and last interaction.
split(interactions[, collect_fold_stats])	Split interactions into folds and apply filtration to the result.

get_test_fold_borders(*interactions:* Interactions) → List[Tuple[Timestamp, Timestamp]]

Return datetime borders of test folds based on given test fold sizes and last interaction.

Parameters

interactions (Interactions) –

Return type

List[*Tuple*[*Timestamp*, *Timestamp*]]

3.6 Tools

3.6.1 Details of RecTools Tools

See the API documentation for further details on Tools:

rectools.tools.ann.ItemToItemAnnRecommender(...)	Class implements item-to-item ANN recommender.
rectools.tools.ann.UserToItemAnnRecommender(...)	Class implements user to item ANN recommender.

ItemToItemAnnRecommender

class rectools.tools.ann.**ItemToItemAnnRecommender**(*item_vectors: np.ndarray*, *item_id_map: tp.Union[IdMap, tp.Dict[ExternalId, InternalId]]*, *index_top_k: int = 0*, *index_init_params: tp.Optional[tp.Dict[str, str]] = None*, *index_query_time_params: tp.Optional[tp.Dict[str, int]] = None*, *create_index_params: tp.Optional[tp.Dict[str, int]] = None*, *index: tp.Optional[nmslib.FloatIndex] = None*)

Bases: *BaseNmslibRecommender*

Class implements item-to-item ANN recommender.

Parameters

- **item_vectors** (*ndarray*) – Ndarray of item latent features of size `(N, K)`, where *N* is the number of items and *K* is the number of features.
- **item_id_map** (*dict(hashable, int) | rectools.datasets.IdMap*) – Mappings from external item ids to internal item ids used by recommender. Values must be positive integers.
- **index_top_k** (*int, default 0*) – Number of items to return per knn query (in addition to *top_n* passed to *get_item_list_for_user*, *get_item_list_for_user_batch*, *get_item_list_for_item* or *get_item_list_for_item_batch*). In this case nmslib index query. This might be important in order to account for filters. See *self.index.knnQueryBatch* in `_compute_sorted_similar'
- **index_init_params** (*optional(dict(str, str)) | rectools.dataset.IdMap*) – NMSLIB initialization parameters. See nmslib documentation. In case of None defaults to reasonable parameters.
- **index_query_time_params** (*optional(dict(str, int)) | rectools.dataset.IdMap*) – NMSLIB query time parameters. See nmslib documentation. In case of None defaults to reasonable parameters.
- **create_index_params** (*optional(dict(str, int)) | rectools.dataset.IdMap*) – NMSLIB index creation parameters. See nmslib documentation. In case of None defaults to reasonable parameters.
- **index** (*FloatIndex, optional*) – Optional instance of FloatIndex. Exists for outside initialization.

get_item_list_for_item()

Part of public API. Given item id and available item ids, calculates recommendations via index query.

Parameters

- **item_id** (*Hashable*) –
- **top_n** (*int*) –
- **item_available_ids** (*Optional[Union[Sequence[Hashable], ndarray]]*) –

Return type

Union[*Sequence*[*Hashable*], *ndarray*]

get_item_list_for_item_batch()

Part of public API. Does exactly what *get_item_list_for_item*, but for a batch of item ids and available item ids.

Parameters

- **item_ids** (*Union[Sequence[Hashable], ndarray]*) –
- **top_n** (*int*) –
- **item_available_ids** (*Optional[Sequence[Union[Sequence[Hashable], ndarray]]]*) –

Return type

Sequence[*Union*[*Sequence*[*Hashable*], *ndarray*]]

See also:

UserToItemAnnRecommender

Inherited-members

Parameters

- **item_vectors** (*np.ndarray*) –
- **item_id_map** (*tp.Union[IdMap, tp.Dict[ExternalId, InternalId]]*) –
- **index_top_k** (*int*) –
- **index_init_params** (*tp.Optional[tp.Dict[str, str]]*) –
- **index_query_time_params** (*tp.Optional[tp.Dict[str, int]]*) –
- **create_index_params** (*tp.Optional[tp.Dict[str, int]]*) –
- **index** (*tp.Optional[nmslib.FloatIndex]*) –

Methods

fit([verbose])	Create and fit *nmslib* index.
get_item_list_for_item(item_id, top_n[, ...])	Calculate top-n recommendations for a given item id and item list.
get_item_list_for_item_batch(item_ids, top_n)	Calculate top-n recommendations for given item ids and item lists.

get_item_list_for_item(*item_id: Hashable*, *top_n: int*, *item_available_ids: Optional[Union[Sequence[Hashable], ndarray]] = None*) → Union[Sequence[Hashable], ndarray]

Calculate top-n recommendations for a given item id and item list. Item list defines which items are allowed to be recommended.

Parameters

- **item_id** (*hashable*) – Item id used by external systems.
- **top_n** (*int*) – How many items to return.
- **item_available_ids** (*optional(sequence(hashable)), default None*) – List of allowed items. In case of None this function recommends without constraints

Returns
Sorted sequence of external ids.

Return type
sequence(hashable)

get_item_list_for_item_batch(*item_ids: Union[Sequence[Hashable], ndarray]*, *top_n: int*, *item_available_ids: Optional[Sequence[Union[Sequence[Hashable], ndarray]]] = None*) → Sequence[Union[Sequence[Hashable], ndarray]]

Calculate top-n recommendations for given item ids and item lists. Item lists define which items are allowed to be recommended.

Parameters

- **item_ids** (*sequence(hashable)*) – List of user ids used by external systems.
- **top_n** (*int*) – How many items to return.
- **item_available_ids** (*optional(sequence(sequence(hashable))), default None*) – List of lists of allowed items for each item id from item_ids in that exact order. In case of None this function recommends without constraints.

Returns
Sequence of sorted sequences of external ids.

Return type
sequence(sequence(hashable))

UserToItemAnnRecommender

class rectools.tools.ann.**UserToItemAnnRecommender**(*user_vectors: np.ndarray*, *item_vectors: np.ndarray*, *user_id_map: tp.Union[IdMap, tp.Dict[ExternalId, InternalId]]*, *item_id_map: tp.Union[IdMap, tp.Dict[ExternalId, InternalId]]*, *index_top_k: int = 0*, *index_init_params: tp.Optional[tp.Dict[str, str]] = None*, *index_query_time_params: tp.Optional[tp.Dict[str, int]] = None*, *create_index_params: tp.Optional[tp.Dict[str, int]] = None*, *index: tp.Optional[nmslib.FloatIndex] = None*)

Bases: *BaseNmslibRecommender*

Class implements user to item ANN recommender.

Parameters

- **user_vectors** (*ndarray*) – Ndarray of user latent features of size `(M, K)`, where *M* is the number of items and *K* is the number of features.
- **item_vectors** (*ndarray*) – Ndarray of item latent features of size `(N, K)`, where *N* is the number of items and *K* is the number of features.
- **user_id_map** (*dict(hashable, int) | rectools.dataset.IdMap*) – Mappings from external user ids to internal user ids used by recommender. Values must be positive integers.
- **item_id_map** (*dict(hashable, int) | rectools.dataset.IdMap*) – Mappings from external item ids to internal item ids used by recommender. Values must be positive integers.
- **index_top_k** (*int, default 0*) – Number of items to return per knn query (in addition to *top_n* passed to *get_item_list_for_user*, *get_item_list_for_user_batch*, *get_item_list_for_item* or *get_item_list_for_item_batch*). This might be important in order to account for filters. See *self.index.knnQueryBatch* in `_compute_sorted_similar'
- **index_init_params** (*optional(dict(str, str)), default None*) – NMSLIB initialization parameters. See nmslib documentation. In case of None defaults to reasonable parameters.
- **index_query_time_params** (*optional(dict(str, int)), default None*) – NMSLIB query time parameters. See nmslib documentation. In case of None defaults to reasonable parameters.
- **create_index_params** (*optional(dict(str, int)), default None*) – NMSLIB index creation parameters. See nmslib documentation. In case of None defaults to reasonable parameters.
- **index** (*FloatIndex, optional*) – Optional instance of *FloatIndex*. Exists for outside initialization.

`get_item_list_for_user()`

Part of public API. Given user id and item ids, calculates recommendations via index query.

Parameters

- **user_id** (*Hashable*) –
- **top_n** (*int*) –
- **item_ids** (*Optional[Union[Sequence[Hashable], ndarray]]*) –

Return type

Union[*Sequence*[*Hashable*], *ndarray*]

`get_item_list_for_user_batch()`

Part of public API. Does what get_item_list_for_user, except it takes a batch of user ids and a batch of item sets.

Parameters

- **user_ids** (*Union[Sequence[Hashable], ndarray]*) –
- **top_n** (*int*) –
- **item_ids** (*Optional[Sequence[Union[Sequence[Hashable], ndarray]]]*) –

Return type

Sequence[*Union*[*Sequence*[*Hashable*], *ndarray*]]

See also:

ItemToItemAnnRecommender

Inherited-members
Parameters

- **user_vectors** (*np.ndarray*) –
- **item_vectors** (*np.ndarray*) –
- **user_id_map** (*tp.Union[IdMap, tp.Dict[ExternalId, InternalId]]*) –
- **item_id_map** (*tp.Union[IdMap, tp.Dict[ExternalId, InternalId]]*) –
- **index_top_k** (*int*) –
- **index_init_params** (*tp.Optional[tp.Dict[str, str]]*) –
- **index_query_time_params** (*tp.Optional[tp.Dict[str, int]]*) –
- **create_index_params** (*tp.Optional[tp.Dict[str, int]]*) –
- **index** (*tp.Optional[nmslib.FloatIndex]*) –

Methods

fit([verbose])	Create and fit *nmslib* index.
get_item_list_for_user(user_id, top_n[, ...])	Calculate top n recommendations for a given user id.
get_item_list_for_user_batch(user_ids, top_n)	Calculate top-n recommendations for given user ids and item lists.

get_item_list_for_user(*user_id: Hashable*, *top_n: int*, *item_ids: Optional[Union[Sequence[Hashable], ndarray]] = None*) → Union[Sequence[Hashable], ndarray]

Calculate top n recommendations for a given user id.

Parameters

- **user_id** (*hashable*) – User id used by external systems.
- **top_n** (*int*) – How many items to return.
- **item_ids** (*optional(sequence(hashable)), default None*) – A set of item ids from which to recommend. In case of None this function recommends without constraints.

Returns
Sorted sequence of external ids

Return type
sequence(hashable)

get_item_list_for_user_batch(*user_ids: Union[Sequence[Hashable], ndarray]*, *top_n: int*, *item_ids: Optional[Sequence[Union[Sequence[Hashable], ndarray]]] = None*) → Sequence[Union[Sequence[Hashable], ndarray]]

Calculate top-n recommendations for given user ids and item lists. Item lists define which items are allowed to be recommended.

Parameters

- **user_ids** (*sequence(hashable)*) – List of user ids used by external systems.

- **top_n** (*int*) – How many items to return.
- **item_ids** (*optional(sequence(sequence(hashable))), default None*) – List of lists of allowed items for each user id from user_ids in that exact order. In case of None this function recommends without constraints.

Returns
Sequence of sorted sequences of external ids.

Return type
sequence(sequence(hashable))

3.7 Visuals

3.7.1 Details of RecTools Visuals

See the API documentation for further details on Visuals:

rectools.visuals.visual_app.ItemToItemVisualApp(...)	Jupyter widgets app for item-to-item recommendations visualization and models comparison.
rectools.visuals.visual_app.VisualApp(...[, ...])	Jupyter widgets app for user-to-item recommendations visualization and models comparison.

ItemToItemVisualApp

class rectools.visuals.visual_app.**ItemToItemVisualApp**(*data_storage: AppDataStorage*, *auto_display: bool = True*, *formatters: Optional[Dict[str, Callable]] = None*, *rows_limit: int = 20*, *min_width: int = 50*)

Bases: *VisualAppBase*

Jupyter widgets app for item-to-item recommendations visualization and models comparison. Do not create instances of this class directly. Use *ItemToItemVisualApp.construct* method instead.

Inherited-members
Parameters

- **data_storage** (AppDataStorage) –
- **auto_display** (*bool*) –
- **formatters** (*Optional[Dict[str, Callable]]*) –
- **rows_limit** (*int*) –
- **min_width** (*int*) –

Methods

construct(reco, item_data[, selected_items, ...])	Construct visualization widgets for item-to-item recommendations.
display()	Display full VisualApp widget
load(folder_name[, auto_display, ...])	Create widgets from data that was processed and saved earlier.
save(folder_name[, overwrite])	Save stored data to re-create widgets when necessary.

classmethod construct(*reco: Union[DataFrame, Dict[Hashable, DataFrame]]*, *item_data: DataFrame*, *selected_items: Optional[Dict[Hashable, Hashable]] = None*, *n_random_items: int = 0*, *auto_display: bool = True*, *formatters: Optional[Dict[str, Callable]] = None*, *rows_limit: int = 20*, *min_width: int = 100*) → *ItemToItemVisualApp*

Construct visualization widgets for item-to-item recommendations.

This will process raw data and create Jupyter widgets for visual analysis and comparison of different models. Created app outputs both target item data and recommended items data from different models for all of the selected items.

Model names for comparison will be listed from the *reco* dictionary keys or *reco* dataframe *Columns.Model* values depending on the format provided.

Target items for comparison can be predefined or random. For predefined items pass *selected_items* dict with item "names" as keys and item ids as values. For random target items pass *n_random_items* number greater then 0. You must specify at least one of the above or provide both.

Optionally provide *formatters* to process dataframe columns values to desired html outputs.

Parameters

- **reco** (*tp.Union[pd.DataFrame, TablesDict]*) – Recommendations from different models in a form of a pd.DataFrame or a dict. In the dict form model names are supposed to be dict keys, and recommendations from different models are supposed to be pd.DataFrames as dict values. In the DataFrame form all recommendations must be specified in one DataFrame with *Columns.Model* column to separate different models. Other required columns for both forms are:
  - *Columns.TargetItem* - target item id
  - *Columns.Item* - recommended item id
  - Any other columns that you wish to display in widgets (e.g. rank or score)

  The original order of the rows will be preserved. Keep in mind to sort the rows correctly before visualizing. The most intuitive way is to sort by rank in ascending order.
- **item_data** (*pd.DataFrame*) – Data for items that is used for visualisation in both interactions and recommendations widgets. Supposed to be in form of a pandas DataFrame with columns:
  - *Columns.Item* - item id
  - Any other columns with item data (e.g. name, category, popularity, image link)
- **selected_items** (tp.Optional[tp.Dict[tp.Hashable, ExternalId]], default None) – Predefined items that will be displayed in widgets. Item names must be specified as keys of the dict and item ids as values of the dict. Must be provided if *n_random_items* = 0.

- **n_random_items** (*int, default 0*) – Number of random items to add for visualization from target items in recommendation tables. Must be greater then 0 if *selected_items* are not specified.
- **auto_display** (bool, optional, default `True`) – Display widgets right after initialization.
- **formatters** (tp.Optional[tp.Dict[str, tp.Callable]], optional, default `None`) – Formatter functions to apply to columns elements in the sections of interactions and recommendations. Keys of the dict must be columns names (item_data, interactions and recommendations columns can be specified here). Values bust be functions that will be applied to corresponding columns elements. The result of each function must be a unicode string that represents html code. Formatters can be used to format text, create links and display images with html.
- **rows_limit** (*int, optional, default 20*) – Maximum number of rows to display in the sections of interactions and recommendations.
- **min_width** (*int, optional, default 100*) – Minimum column width in pixels for dataframe columns in widgets output. Must be greater then 10.

Return type
ItemToItemVisualApp

Examples

Providing reco as TablesDict

```
>>> reco = {
...     "model_1": pd.DataFrame({Columns.TargetItem: [1, 2], Columns.Item: [3,␣
↪4], Columns.Score: [0.99, 0.9]}),
...     "model_2": pd.DataFrame({Columns.TargetItem: [1, 2], Columns.Item: [5,␣
↪6], Columns.Rank: [1, 1]})
... }
>>>
>>> item_data = pd.DataFrame({
...     Columns.Item: [3, 4, 5, 6, 1, 2],
...     "feature_1": ["one", "two", "three", "five", "one", "two"]
... })
>>>
>>> selected_items = {"item_one": 1}
>>>
>>> app = ItemToItemVisualApp.construct(
...     reco=reco,
...     item_data=item_data,
...     selected_items=selected_items,
...     auto_display=False
... )
```

Providing reco as pd.DataFrame and adding *formatters*

```
>>> reco = pd.DataFrame({
...     Columns.TargetItem: [1, 2, 1, 2],
...     Columns.Item: [3, 4, 5, 6],
...     Columns.Model: ["model_1", "model_1", "model_2", "model_2"]
```

(continues on next page)

(continued from previous page)

```
... })
>>>
>>> item_data = pd.DataFrame({
...     Columns.Item: [3, 4, 5, 6, 1, 2],
...     "feature_1": ["one", "two", "three", "five", "one", "two"]
... })
>>>
>>> selected_items = {"item_one": 1}
>>> formatters = {"item_id": lambda x: f"<b>{x}</b>"}
>>>
>>> app = ItemToItemVisualApp.construct(
...     reco=reco,
...     item_data=item_data,
...     selected_items=selected_items,
...     formatters=formatters,
...     auto_display=False
... )
```

VisualApp

class rectools.visuals.visual_app.**VisualApp**(*data_storage: AppDataStorage*, *auto_display: bool = True*, *formatters: Optional[Dict[str, Callable]] = None*, *rows_limit: int = 20*, *min_width: int = 50*)

Bases: *VisualAppBase*

Jupyter widgets app for user-to-item recommendations visualization and models comparison. Do not create instances of this class directly. Use *VisualApp.construct* method instead.

Inherited-members

Parameters

- **data_storage** (*AppDataStorage*) –
- **auto_display** (*bool*) –
- **formatters** (*Optional[Dict[str, Callable]]*) –
- **rows_limit** (*int*) –
- **min_width** (*int*) –

Methods

construct(reco, interactions, item_data[, ...])	Construct visualization app for classic user-to-item recommendations.
display()	Display full VisualApp widget
load(folder_name[, auto_display, ...])	Create widgets from data that was processed and saved earlier.
save(folder_name[, overwrite])	Save stored data to re-create widgets when necessary.

classmethod construct(*reco: Union[DataFrame, Dict[Hashable, DataFrame]]*, *interactions: DataFrame*, *item_data: DataFrame*, *selected_users: Optional[Dict[Hashable, Hashable]] = None*, *n_random_users: int = 0*, *auto_display: bool = True*, *formatters: Optional[Dict[str, Callable]] = None*, *rows_limit: int = 20*, *min_width: int = 100*) → *VisualApp*

Construct visualization app for classic user-to-item recommendations.

This will process raw data and create Jupyter widgets for visual analysis and comparison of different models. Created app outputs both interactions history of the selected users and their recommended items from different models along with explicit items data.

Model names for comparison will be listed from the *reco* dictionary keys or *reco* dataframe *Columns.Model* values depending on the format provided.

Users for comparison can be predefined or random. For predefined users pass *selected_users* dict with user "names" as keys and user ids as values. For random users pass *n_random_users* number greater then 0. You must specify at least one of the above or provide both.

Optionally provide *formatters* to process dataframe columns values to desired html outputs.

Parameters

- **reco** (*tp.Union[pd.DataFrame, TablesDict]*) – Recommendations from different models in a form of a pd.DataFrame or a dict. In the dict form model names are supposed to be dict keys, and recommendations from different models are supposed to be pd.DataFrames as dict values. In the DataFrame form all recommendations must be specified in one DataFrame with *Columns.Model* column to separate different models. Other required columns for both forms are:
  - *Columns.User* - user id
  - *Columns.Item* - recommended item id
  - Any other columns that you wish to display in widgets (e.g. rank or score)

  The original order of the rows will be preserved. Keep in mind to sort the rows correctly before visualizing. The most intuitive way is to sort by rank in ascending order.
- **interactions** (*pd.DataFrame*) – Table with interactions history for users. Only needed for u2i case. Supposed to be in form of pandas DataFrames with columns:
  - *Columns.User* - user id
  - *Columns.Item* - item id

  The original order of the rows will be preserved. Keep in mind to sort the rows correctly before visualizing. The most intuitive way is to sort by date in descending order. If user has too many interactions the lest ones may not be displayed.
- **item_data** (*pd.DataFrame*) – Data for items that is used for visualisation in both interactions and recommendations widgets. Supposed to be in form of a pandas DataFrame with columns:
  - *Columns.Item* - item id
  - Any other columns with item data (e.g. name, category, popularity, image link)
- **selected_users** (tp.Optional[tp.Dict[tp.Hashable, ExternalId]], default `None`) – Predefined users that will be displayed in widgets. User names must be specified as keys of the dict and user ids as values of the dict. Must be provided if *n_random_users* = 0.

- **n_random_users** (*int, default 0*) – Number of random users to add for visualization from users in recommendation tables. Must be greater then 0 if *selected_users* are not specified.
- **auto_display** (bool, optional, default `True`) – Display widgets right after initialization.
- **formatters** (tp.Optional[tp.Dict[str, tp.Callable]], optional, default `None`) – Formatter functions to apply to columns elements in the sections of interactions and recommendations. Keys of the dict must be columns names (item_data, interactions and recommendations columns can be specified here). Values bust be functions that will be applied to corresponding columns elements. The result of each function must be a unicode string that represents html code. Formatters can be used to format text, create links and display images with html.
- **rows_limit** (*int, optional, default 20*) – Maximum number of rows to display in the sections of interactions and recommendations.
- **min_width** (*int, optional, default 100*) – Minimum column width in pixels for dataframe columns in widgets output. Must be greater then 10.

Return type
VisualApp

Examples

Providing reco as TablesDict

```
>>> reco = {
...     "model_1": pd.DataFrame({Columns.User: [1, 2], Columns.Item: [3, 4],␣
↪Columns.Score: [0.99, 0.9]}),
...     "model_2": pd.DataFrame({Columns.User: [1, 2], Columns.Item: [5, 6],␣
↪Columns.Rank: [1, 1]})
... }
>>>
>>> item_data = pd.DataFrame({
...     Columns.Item: [3, 4, 5, 6, 7, 8],
...     "feature_1": ["one", "two", "three", "five", "one", "two"]
... })
>>>
>>> interactions = pd.DataFrame({Columns.User: [1, 1, 2], Columns.Item: [3, 7,␣
↪8]})
>>> selected_users = {"user_one": 1}
>>>
>>> app = VisualApp.construct(
...     reco=reco,
...     item_data=item_data,
...     interactions=interactions,
...     selected_users=selected_users,
...     auto_display=False
... )
```

Providing reco as pd.DataFrame and adding *formatters*

```
>>> reco = pd.DataFrame({
...     Columns.User: [1, 2, 1, 2],
...     Columns.Item: [3, 4, 5, 6],
...     Columns.Model: ["model_1", "model_1", "model_2", "model_2"]
... })
>>>
>>> item_data = pd.DataFrame({
...     Columns.Item: [3, 4, 5, 6, 7, 8],
...     "feature_1": ["one", "two", "three", "five", "one", "two"]
... })
>>>
>>> interactions = pd.DataFrame({Columns.User: [1, 1, 2], Columns.Item: [3, 7,␣
↪8]})
>>> selected_users = {"user_one": 1}
>>>
>>> formatters = {"item_id": lambda x: f"<b>{x}</b>"}
>>>
>>> app = VisualApp.construct(
...     reco=reco,
...     item_data=item_data,
...     interactions=interactions,
...     selected_users=selected_users,
...     formatters=formatters,
...     auto_display=False
... )
```

3.8 API

rectools.dataset	Data conversion tools (rectools.dataset).
rectools.metrics	Metrics calculation tools (rectools.metrics).
rectools.model_selection	Model selection tools (rectools.model_selection)
rectools.models	Recommendation models (rectools.models)
rectools.tools	Tools (rectools.tools)
rectools.visuals	Visualization tools (rectools.visuals)

3.8.1 dataset

Data conversion tools (`rectools.dataset`).

Data and identifiers conversion tools for future working with models.

Data Containers

dataset.IdMap - Mapping between external and internal identifiers. *dataset.DenseFeatures* - Container for dense features. *dataset.SparseFeatures* - Container for sparse features. *dataset.Interactions* - Container for interactions. *dataset.Dataset* - Container for all data.

Modules

rectools.dataset.dataset	Dataset - all data container.
rectools.dataset.features	Structures to save explicit features.
rectools.dataset.identifiers	Mapping between external and internal ids.
rectools.dataset.interactions	Structure for saving user-item interactions.
rectools.dataset.torch_datasets	Special datasets used in neural models.

dataset

Dataset - all data container.

Classes

Dataset(user_id_map, item_id_map, interactions)	Container class for all data for a recommendation model.

features

Structures to save explicit features.

Classes

DenseFeatures(values, names)	Storage for dense features.
SparseFeatures(values, names)	Storage for sparse features.

Exceptions

AbsentIdError	The error is raised when there are some ids in the id map that are not present in the dataframe
UnknownIdError	The error is raised when there are some ids in the dataframe that are not present in the id map

rectools.dataset.features.AbsentIdError

exception rectools.dataset.features.**AbsentIdError**

The error is raised when there are some ids in the id map that are not present in the dataframe

rectools.dataset.features.UnknownIdError

exception rectools.dataset.features.**UnknownIdError**

The error is raised when there are some ids in the dataframe that are not present in the id map

identifiers

Mapping between external and internal ids.

Classes

IdMap(external_ids)	Mapping between external and internal object ids.

interactions

Structure for saving user-item interactions.

Classes

Interactions(df)	Structure to store info about user-item interactions.

torch_datasets

Special datasets used in neural models.

Classes

DSSMItemDataset(items)	Torch dataset wrapper for *rectools.dataset.dataset.Dataset*.
DSSMItemDatasetBase(*args, **kwargs)	Base class for DSSM item datasets.
DSSMTrainDataset(items, users, interactions)	Torch dataset wrapper for *rectools.dataset.dataset.Dataset*.
DSSMTrainDatasetBase(*args, **kwargs)	Base class for DSSM training datasets.
DSSMUserDataset(users, interactions[, ...])	Torch dataset wrapper for *rectools.dataset.dataset.Dataset*.
DSSMUserDatasetBase(*args, **kwargs)	Base class for DSSM training datasets.

DSSMItemDataset

class rectools.dataset.torch_datasets.**DSSMItemDataset**(*items: csr_matrix*)

Bases: DSSMItemDatasetBase

Torch dataset wrapper for *rectools.dataset.dataset.Dataset*. Implements *torch.utils.data.Dataset* for subsequent usage with *torch.utils.data.DataLoader*. Does the following: for a given index takes a row of item features and then returns them as tensors.

This class is intended for internal usage or advanced users.

Inherited-members

Methods

from_dataset(dataset)

DSSMItemDatasetBase

class rectools.dataset.torch_datasets.**DSSMItemDatasetBase**(**args: Any*, ***kwargs: Any*)

Bases: Dataset[Any]

Base class for DSSM item datasets. Used only for type hinting.

Inherited-members

Methods

from_dataset(dataset)

DSSMTrainDataset

class rectools.dataset.torch_datasets.**DSSMTrainDataset**(*items: csr_matrix*, *users: csr_matrix*, *interactions: csr_matrix*)

Bases: *DSSMTrainDatasetBase*

Torch dataset wrapper for *rectools.dataset.dataset.Dataset*. Implements *torch.utils.data.Dataset* for subsequent usage with *torch.utils.data.DataLoader*. Does the following: for a given index takes a row of user interactions, a row of user features and samples one positive and one negative items and then returns them as tensors.

This class is intended for internal usage or advanced users who want to implement more sophisticated sampling logic.

Parameters

- **items** (*csr_matrix*) – Item features.
- **users** (*csr_matrix*) – User features.
- **interactions** (*csr_matrix*) – Interactions matrix.

Inherited-members

Methods

from_dataset(dataset)

DSSMTrainDatasetBase

class rectools.dataset.torch_datasets.**DSSMTrainDatasetBase**(**args: Any*, ***kwargs: Any*)

Bases: Dataset[Any]

Base class for DSSM training datasets. Used only for type hinting.

Inherited-members

Methods

from_dataset(dataset)

DSSMUserDataset

class rectools.dataset.torch_datasets.**DSSMUserDataset**(*users: csr_matrix*, *interactions: csr_matrix*, *keep_users: Optional[Sequence[int]] = None*)

Bases: *DSSMUserDatasetBase*

Torch dataset wrapper for *rectools.dataset.dataset.Dataset*. Implements *torch.utils.data.Dataset* for subsequent usage with *torch.utils.data.DataLoader*. Does the following: for a given index takes a row of user interactions, a row of user features and then returns them as tensors.

This class is intended for internal usage or advanced users.

Inherited-members

Methods

`from_dataset`(dataset[, keep_users])

DSSMUserDatasetBase

class rectools.dataset.torch_datasets.**DSSMUserDatasetBase**(**args: Any*, ***kwargs: Any*)

Bases: `Dataset[Any]`

Base class for DSSM training datasets. Used only for type hinting.

Inherited-members

Methods

`from_dataset`(dataset[, keep_users])

3.8.2 metrics

Metrics calculation tools (`rectools.metrics`).

Tools for fast and convenient calculation of different recommendation metrics.

Metrics

metrics.Precision metrics.Recall metrics.F1Beta metrics.Accuracy metrics.MCC metrics.MAP metrics.NDCG metrics.MRR metrics.MeanInvUserFreq metrics.IntraListDiversity metrics.AvgRecPopularity metrics.Serendipity metrics.HitRate

Tools

metrics.calc_metrics - calculate a set of metrics efficiently *metrics.PairwiseDistanceCalculator metrics.PairwiseHammingDistanceCalculator metrics.SparsePairwiseHammingDistanceCalculator*

Modules

rectools.metrics.base	Base metric module.
rectools.metrics.classification	Classification recommendations metrics.
rectools.metrics.distances	Distance metrics.
rectools.metrics.diversity	Diversity metrics.
rectools.metrics.novelty	Novelty metrics.
rectools.metrics.popularity	Popularity metrics.
rectools.metrics.ranking	Ranking recommendations metrics.
rectools.metrics.scoring	Metrics calculation module.
rectools.metrics.serendipity	Serendipity is designed to find balance between novelty and relevance.

base

Base metric module.

Functions

merge_reco(reco, interactions)	Merge recommendation table with interactions table.

merge_reco

rectools.metrics.base.**merge_reco**(*reco: DataFrame*, *interactions: DataFrame*) → DataFrame

Merge recommendation table with interactions table.

Parameters

- **reco** (*pd.DataFrame*) – Recommendations table with columns *Columns.User*, *Columns.Item*, *Columns.Rank*.
- **interactions** (*pd.DataFrame*) – Interactions table with columns *Columns.User*, *Columns.Item*.

Returns

Result of merging.

Return type

pd.DataFrame

Classes

MetricAtK(k)	Base class of metrics that depends on k - a number of top recommendations used to calculate a metric.

MetricAtK

class rectools.metrics.base.**MetricAtK**(*k: int*)

Bases: object

Base class of metrics that depends on k - a number of top recommendations used to calculate a metric.

Warning: This class should not be used directly. Use derived classes instead.

Parameters

k (*int*) – Number of items at the top of recommendations list that will be used to calculate metric.

Inherited-members

Attributes

k

classification

Classification recommendations metrics.

Functions

calc_classification_metrics(metrics, merged)	Calculate any classification metrics.
calc_confusions(merged, k)	Calculate some intermediate metrics from prepared data (it's a helper function).
make_confusions(reco, interactions, k)	Calculate some intermediate metrics from raw data (it's a helper function).

calc_classification_metrics

rectools.metrics.classification.**calc_classification_metrics**(*metrics: Dict[str, Union[ClassificationMetric, SimpleClassificationMetric]], merged: DataFrame, catalog: Optional[Collection[Union[str, int]]] = None*) → Dict[str, float]

Calculate any classification metrics.

Works with prepared data.

Warning: It is not recommended to use this function directly. Use *calc_metrics* instead.

Parameters

- **metrics** (*dict(str -> (ClassificationMetric | SimpleClassificationMetric))*) – Dict of metric objects to calculate, where key is a metric name and value is a metric object.

- **merged** (*pd.DataFrame*) – Result of merging recommendations and interactions tables. Can be obtained using *merge_reco* function.
- **catalog** (*collection, optional*) – Collection of unique item ids that could be used for recommendations. Obligatory only if *metrics* contains *ClassificationMetric* instances.

Returns
Dictionary where keys are the same as keys in *metrics* and values are metric calculation results.

Return type
dict(str->float)

Raises

- **ValueError** – If *n_items* is not passed and *ClassificationMetric* is present in *metrics*.
- **TypeError** – If unexpected metric is present in *metrics*.

calc_confusions

rectools.metrics.classification.**calc_confusions**(*merged: DataFrame*, *k: int*) → DataFrame

Calculate some intermediate metrics from prepared data (it's a helper function).

For each user (*Columns.User*) the following metrics are calculated:

- *LIKED* - number of items the user has interacted (bought, liked) with;
- *TP* - number of relevant recommendations among the first *k* items at the top of recommendation list;
- *FP* - number of non-relevant recommendations among the first *k* items of recommendation list;
- *FN* - number of items the user has interacted with but that weren't recommended (in top *k*).

Parameters

- **merged** (*pd.DataFrame*) – Result of merging recommendations and interactions tables. Can be obtained using *merge_reco* function.
- **k** (*int*) – Number of items at the top of recommendations list that will be used to calculate metric.

Returns
Table with columns: *Columns.User*, *LIKED*, *TP*, *FP*, *FN*.

Return type
pd.DataFrame

Notes

left = all - K TP = sum(rank) FP = K - TP FN = liked - TP TN = all - K - FN = left - FN = left - liked + TP

make_confusions

rectools.metrics.classification.**make_confusions**(*reco: DataFrame*, *interactions: DataFrame*, *k: int*) → DataFrame

Calculate some intermediate metrics from raw data (it's a helper function).

For each user the following metrics are calculated:

- *LIKED* - number of items the user has interacted (bought, liked) with;
- *TP* - number of relevant recommendations among the first *k* items at the top of recommendation list;
- *FP* - number of non-relevant recommendations among the first *k* items of recommendation list;
- *FN* - number of items the user has interacted with but that weren't recommended (in top-*k*).

Parameters

- **reco** (*pd.DataFrame*) – Recommendations table with columns *Columns.User*, *Columns.Item*, *Columns.Rank*.
- **interactions** (*pd.DataFrame*) – Interactions table with columns *Columns.User*, *Columns.Item*.
- **k** (*int*) – Number of items at the top of recommendations list that will be used to calculate metric.

Returns

Table with columns: *Columns.User*, *LIKED*, *TP*, *FP*, *FN*.

Return type

pd.DataFrame

Classes

Accuracy(k)	Ratio of correctly recommended items among all items.
ClassificationMetric(k)	Classification metric base class.
F1Beta(k[, beta])	Fbeta score for k first recommendations.
HitRate(k)	HitRate calculates the fraction of users for which the correct answer is included in the recommendation list.
MCC(k)	Matthew correlation coefficient calculates correlation between actual and predicted classification.
Precision(k)	Ratio of relevant items among top-*k* recommended items.
Recall(k)	Ratio of relevant recommended items among all items user interacted with after recommendations were made.
SimpleClassificationMetric(k)	Simple classification metric base class.

ClassificationMetric

class rectools.metrics.classification.**ClassificationMetric**(*k: int*)

Bases: *MetricAtK*

Classification metric base class.

Warning: This class should not be used directly. Use derived classes instead.

Parameters

k (*int*) – Number of items at the top of recommendations list that will be used to calculate metric.

Inherited-members

Methods

calc(reco, interactions, catalog)	Calculate metric value.
calc_from_confusion_df(confusion_df, catalog)	Calculate metric value from prepared confusion matrix.
calc_per_user(reco, interactions, catalog)	Calculate metric values for all users.
calc_per_user_from_confusion_df(...)	Calculate metric values for all users from prepared confusion matrix.

Attributes

calc(*reco: DataFrame*, *interactions: DataFrame*, *catalog: Collection[Union[str, int]]*) → float

Calculate metric value.

Parameters

- **reco** (*pd.DataFrame*) – Recommendations table with columns *Columns.User*, *Columns.Item*, *Columns.Rank*.
- **interactions** (*pd.DataFrame*) – Interactions table with columns *Columns.User*, *Columns.Item*.
- **catalog** (*collection*) – Collection of unique item ids that could be used for recommendations.

Returns

Value of metric (average between users).

Return type

float

calc_from_confusion_df(*confusion_df: DataFrame*, *catalog: Collection[Union[str, int]]*) → float

Calculate metric value from prepared confusion matrix.

Parameters

- **confusion_df** (*pd.DataFrame*) – Table with some confusion values for every user. Columns are: *Columns.User*, *LIKED*, *TP*, *FP*, *FN*. This table can be generated by *make_confusions* (or *calc_confusions*) function. See its description for details.

- **catalog** (*collection*) – Collection of unique item ids that could be used for recommendations.

Returns
Value of metric (average between users).

Return type
float

calc_per_user(*reco: DataFrame*, *interactions: DataFrame*, *catalog: Collection[Union[str, int]]*) → Series

Calculate metric values for all users.

Parameters

- **reco** (*pd.DataFrame*) – Recommendations table with columns *Columns.User*, *Columns.Item*, *Columns.Rank*.
- **interactions** (*pd.DataFrame*) – Interactions table with columns *Columns.User*, *Columns.Item*.
- **catalog** (*collection*) – Collection of unique item ids that could be used for recommendations.

Returns
Values of metric (index - user id, values - metric value for every user).

Return type
pd.Series

calc_per_user_from_confusion_df(*confusion_df: DataFrame*, *catalog: Collection[Union[str, int]]*) → Series

Calculate metric values for all users from prepared confusion matrix.

Parameters

- **confusion_df** (*pd.DataFrame*) – Table with some confusion values for every user. Columns are: *Columns.User*, *LIKED*, *TP*, *FP*, *FN*. This table can be generated by *make_confusions* (or *calc_confusions*) function. See its description for details.
- **catalog** (*collection*) – Collection of unique item ids that could be used for recommendations.

Returns
Values of metric (index - user id, values - metric value for every user).

Return type
pd.Series

SimpleClassificationMetric

class rectools.metrics.classification.**SimpleClassificationMetric**(*k: int*)

Bases: *MetricAtK*

Simple classification metric base class.

Warning: This class should not be used directly. Use derived classes instead.

Parameters
k (*int*) – Number of items at the top of recommendations list that will be used to calculate metric.

Inherited-members

Methods

calc(reco, interactions)	Calculate metric value.
calc_from_confusion_df(confusion_df)	Calculate metric value from prepared confusion matrix.
calc_per_user(reco, interactions)	Calculate metric values for all users.
calc_per_user_from_confusion_df(confusion_df)	Calculate metric values for all users from prepared confusion matrix.

Attributes

calc(*reco: DataFrame*, *interactions: DataFrame*) → float

Calculate metric value.

Parameters

- **reco** (*pd.DataFrame*) – Recommendations table with columns *Columns.User*, *Columns.Item*, *Columns.Rank*.
- **interactions** (*pd.DataFrame*) – Interactions table with columns *Columns.User*, *Columns.Item*.

Returns
Value of metric (average between users).

Return type
float

calc_from_confusion_df(*confusion_df: DataFrame*) → float

Calculate metric value from prepared confusion matrix.

Parameters
confusion_df (*pd.DataFrame*) – Table with some confusion values for every user. Columns are: *Columns.User*, *LIKED*, *TP*, *FP*, *FN*. This table can be generated by *make_confusions* (or *calc_confusions*) function. See its description for details.

Returns
Value of metric (average between users).

Return type
float

calc_per_user(*reco: DataFrame*, *interactions: DataFrame*) → Series

Calculate metric values for all users.

Parameters

- **reco** (*pd.DataFrame*) – Recommendations table with columns *Columns.User*, *Columns.Item*, *Columns.Rank*.
- **interactions** (*pd.DataFrame*) – Interactions table with columns *Columns.User*, *Columns.Item*.

Returns
Values of metric (index - user id, values - metric value for every user).

Return type
pd.Series

`calc_per_user_from_confusion_df`(*confusion_df: DataFrame*) → Series

Calculate metric values for all users from prepared confusion matrix.

Parameters
confusion_df (*pd.DataFrame*) – Table with some confusion values for every user. Columns are: *Columns.User*, *LIKED*, *TP*, *FP*, *FN*. This table can be generated by *make_confusions* (or *calc_confusions*) function. See its description for details.

Returns
Values of metric (index - user id, values - metric value for every user).

Return type
pd.Series

distances

Distance metrics.

Classes

PairwiseDistanceCalculator()	Base pairwise distance calculator class
PairwiseHammingDistanceCalculator(...)	Class for computing Hamming distance between a pair of items.
SparsePairwiseHammingDistanceCalculator(...)	Class for computing Hamming distance between multiple pairs of elements represented in features matrix in sparse form.

diversity

Diversity metrics.

Functions

calc_diversity_metrics(metrics, reco)	Calculate diversity metrics (only IntraListDiversity now).

calc_diversity_metrics

rectools.metrics.diversity.**calc_diversity_metrics**(*metrics: Dict[str, IntraListDiversity]*, *reco: DataFrame*) → Dict[str, float]

Calculate diversity metrics (only IntraListDiversity now).

Warning: It is not recommended to use this function directly. Use *calc_metrics* instead.

Parameters

- **metrics** (*dict(str -> DiversityMetric)*) – Dict of metric objects to calculate, where key is metric name and value is metric object.

- **reco** (*pd.DataFrame*) – Recommendations table with columns *Columns.User*, *Columns.Item*, *Columns.Rank*.

Returns

Dictionary where keys are the same with keys in *metrics* and values are metric calculation results.

Return type

dict(str->float)

Classes

DiversityMetric	alias of *IntraListDiversity*
ILDFitted(recommended_items_paired, users)	Container with meta data got from *IntraListDiversity.fit* method.
IntraListDiversity(k, distance_calculator)	Intra-List Diversity metric.

DiversityMetric

rectools.metrics.diversity.**DiversityMetric**

alias of *IntraListDiversity*

ILDFitted

class rectools.metrics.diversity.**ILDFitted**(*recommended_items_paired: DataFrame*, *users: ndarray*)

Bases: object

Container with meta data got from *IntraListDiversity.fit* method.

Parameters

- **recommended_items_paired** (*pd.DataFrame*) – Table with recommended item pairs, with columns item_0, item_1, rank_0, rank_1.
- **users** (*np.ndarray*) – Array of user ids.

Inherited-members

Attributes

recommended_items_paired
users

novelty

Novelty metrics.

Functions

calc_novelty_metrics(metrics, reco, ...)	Calculate novelty metrics (only MeanInvUserFreq now).

calc_novelty_metrics

rectools.metrics.novelty.**calc_novelty_metrics**(*metrics: Dict[str, MeanInvUserFreq]*, *reco: DataFrame*, *prev_interactions: DataFrame*) → Dict[str, float]

Calculate novelty metrics (only MeanInvUserFreq now).

Warning: It is not recommended to use this function directly. Use *calc_metrics* instead.

Parameters

- **metrics** (*dict(str -> NoveltyMetric)*) – Dict of metric objects to calculate, where key is metric name and value is metric object.
- **reco** (*pd.DataFrame*) – Recommendations table with columns *Columns.User*, *Columns.Item*, *Columns.Rank*.
- **prev_interactions** (*pd.DataFrame*) – Table with previous user-item interactions, with columns *Columns.User*, *Columns.Item*.

Returns

Dictionary where keys are the same as keys in *metrics* and values are metric calculation results.

Return type

dict(str->float)

Classes

MIUFFitted(item_novelties, users)	Container with meta data got from *MeanInvUserFreq.fit* method.
MeanInvUserFreq(k)	Mean Inverse User Frequency metric.
NoveltyMetric	alias of *MeanInvUserFreq*

MIUFFitted

class rectools.metrics.novelty.**MIUFFitted**(*item_novelties: DataFrame*, *users: ndarray*)

Bases: object

Container with meta data got from *MeanInvUserFreq.fit* method.

Parameters

- **item_novelties** (*pd.DataFrame*) – Table with columns *Columns.User*, *Columns.Item*, item_novelty.
- **users** (*np.ndarray*) – Array of user ids.

Inherited-members

Attributes

item_novelties
users

NoveltyMetric

rectools.metrics.novelty.**NoveltyMetric**

alias of *MeanInvUserFreq*

popularity

Popularity metrics.

Functions

calc_popularity_metrics(metrics, reco, ...)	Calculate popularity metrics (only AvgRP now).

calc_popularity_metrics

rectools.metrics.popularity.**calc_popularity_metrics**(*metrics: Dict[str, AvgRecPopularity]*, *reco: DataFrame*, *prev_interactions: DataFrame*) → Dict[str, float]

Calculate popularity metrics (only AvgRP now).

Warning: It is not recommended to use this function directly. Use *calc_metrics* instead.

Parameters

- **metrics** (*dict(str -> PopularityMetric)*) – Dict of metric objects to calculate, where key is metric name and value is metric object.
- **reco** (*pd.DataFrame*) – Recommendations table with columns *Columns.User*, *Columns.Item*, *Columns.Rank*.
- **prev_interactions** (*pd.DataFrame*) – Table with previous user-item interactions, with columns *Columns.User*, *Columns.Item*.

Returns

Dictionary where keys are the same as keys in *metrics* and values are metric calculation results.

Return type

dict(str->float)

Classes

AvgRecPopularity(k[, normalize])	Average Recommendations Popularity metric.
PopularityMetric	alias of *AvgRecPopularity*

PopularityMetric

rectools.metrics.popularity.**PopularityMetric**

alias of *AvgRecPopularity*

ranking

Ranking recommendations metrics.

Functions

calc_ranking_metrics(metrics, merged)	Calculate any ranking metrics (MAP, NDCG and MRR for now).

calc_ranking_metrics

rectools.metrics.ranking.**calc_ranking_metrics**(*metrics: Dict[str, Union[*NDCG, MAP, MRR*]]*, *merged: DataFrame*) → Dict[str, float]

Calculate any ranking metrics (MAP, NDCG and MRR for now).

Works with pre-prepared data.

Warning: It is not recommended to use this function directly. Use *calc_metrics* instead.

Parameters

- **metrics** (*dict(str -> (*MAP | NDCG | MRR*))*) – Dict of metric objects to calculate, where key is metric name and value is metric object.
- **merged** (*pd.DataFrame*) – Result of merging recommendations and interactions tables. Can be obtained using *merge_reco* function.

Returns

Dictionary where keys are the same with keys in *metrics* and values are metric calculation results.

Return type

dict(str->float)

Classes

MAP(k[, divide_by_k])	Mean Average Precision at k (MAP@k).
MAPFitted(precision_at_k, users, ...)	Container with meta data got from *MAP.fit* method.
MRR(k)	Mean Reciprocal Rank at k (MRR@k).
NDCG(k[, log_base])	Normalized Discounted Cumulative Gain at k (NDCG@k).
_RankingMetric(k)	Simple classification metric base class.

MAPFitted

class rectools.metrics.ranking.**MAPFitted**(*precision_at_k: csr_matrix*, *users: ndarray*, *n_relevant_items: ndarray*)

Bases: `object`

Container with meta data got from *MAP.fit* method.

Parameters

- **precision_at_k** (*csr_matrix*) – CSR matrix where rows corresponds to users, rows corresponds all possible k from 0 to *k_max*, and values are weighted precisions for relevant recommended items.
- **users** (*np.ndarray*) – Array of user ids.
- **n_relevant_items** (*np.ndarray*) – Tally of relevant items for each user. Users are in the same order as in *precision_at_k* matrix.

Inherited-members

Attributes

precision_at_k
users
n_relevant_items

_RankingMetric

class rectools.metrics.ranking.**_RankingMetric**(*k: int*)

Bases: *MetricAtK*

Simple classification metric base class.

Warning: This class should not be used directly. Use derived classes instead.

Parameters

k (*int*) – Number of items at the top of recommendations list that will be used to calculate metric.

Inherited-members

Methods

calc(reco, interactions)	Calculate metric value.
calc_per_user(reco, interactions)	Calculate metric values for all users.

Attributes

calc(*reco: DataFrame*, *interactions: DataFrame*) → float

Calculate metric value.

Parameters

- **reco** (*pd.DataFrame*) – Recommendations table with columns *Columns.User*, *Columns.Item*, *Columns.Rank*.
- **interactions** (*pd.DataFrame*) – Interactions table with columns *Columns.User*, *Columns.Item*.

Returns
Value of metric (average between users).

Return type
float

calc_per_user(*reco: DataFrame*, *interactions: DataFrame*) → Series

Calculate metric values for all users.

Parameters

- **reco** (*pd.DataFrame*) – Recommendations table with columns *Columns.User*, *Columns.Item*, *Columns.Rank*.
- **interactions** (*pd.DataFrame*) – Interactions table with columns *Columns.User*, *Columns.Item*.

Returns
Values of metric (index - user id, values - metric value for every user).

Return type
pd.Series

scoring

Metrics calculation module.

Functions

calc_metrics(metrics, reco[, interactions, ...])	Calculate metrics.

serendipity

Serendipity is designed to find balance between novelty and relevance.

Functions

calc_serendipity_metrics(metrics, reco, ...)	Calculate serendipity metrics.

calc_serendipity_metrics

rectools.metrics.serendipity.**calc_serendipity_metrics**(*metrics: Dict[str, Serendipity]*, *reco: DataFrame*, *interactions: DataFrame*, *prev_interactions: DataFrame*, *catalog: Collection[Union[str, int]]*) → Dict[str, float]

Calculate serendipity metrics.

Warning: It is not recommended to use this function directly. Use *calc_metrics* instead.

Parameters

- **metrics** (*dict(str -> SerendipityMetric)*) – Dict of metric objects to calculate, where key is metric name and value is metric object.
- **reco** (*pd.DataFrame*) – Recommendations table with columns *Columns.User*, *Columns.Item*, *Columns.Rank*.
- **interactions** (*pd.DataFrame*) – Interactions table with columns *Columns.User*, *Columns.Item*.
- **prev_interactions** (*pd.DataFrame*) – Table with previous user-item interactions, with columns *Columns.User*, *Columns.Item*.
- **catalog** (*collection*) – Collection of unique item ids that could be used for recommendations.

Returns

Dictionary where keys are the same as keys in *metrics* and values are metric calculation results.

Return type

dict(str->float)

Classes

Serendipity(k)	Serendipity metric.
SerendipityFitted(serendipity_values, users)	Container with meta data got from *Serendipity.fit* method.
SerendipityMetric	alias of *Serendipity*

SerendipityFitted

class rectools.metrics.serendipity.**SerendipityFitted**(*serendipity_values: DataFrame*, *users: ndarray*)

Bases: `object`

Container with meta data got from *Serendipity.fit* method.

Parameters

- **serendipity_values** (*pd.DataFrame*) – Table with serendipity value for every recommended item, with columns *Columns.User*, *Columns.Rank*, `serendipity`,
- **users** (*np.ndarray*) – Array of user ids.

Inherited-members

Attributes

serendipity_values
users

SerendipityMetric

rectools.metrics.serendipity.**SerendipityMetric**

alias of *Serendipity*

3.8.3 model_selection

Model selection tools (`rectools.model_selection`)

Instruments to validate and compare models.

Splitters

model_selection.Splitter - base class for all splitters

model_selection.KFoldSplitter - split interactions randomly *model_selection.LastNSplitter* - split interactions by recent activity *model_selection.TimeRangeSplit* - split interactions by time

Model selection tools

model_selection.cross_validate - run cross validation on multiple models with multiple metrics

Modules

rectools.model_selection.cross_validate(...)	Run cross validation on multiple models with multiple metrics.
rectools.model_selection.last_n_split	LastNSplitter.
rectools.model_selection.random_split	RandomSplitter.
rectools.model_selection.splitter	Splitter.
rectools.model_selection.time_split	TimeRangeSplitter.
rectools.model_selection.utils	

cross_validate

last_n_split

LastNSplitter.

Classes

LastNSplitter(n[, n_splits, ...])	Splitter for cross-validation by leave-one-out / leave-k-out scheme (recent activity).

random_split

RandomSplitter.

Classes

RandomSplitter(test_fold_frac[, n_splits, ...])	Slitter for cross-validation by random.

splitter

Splitter.

Classes

Splitter([filter_cold_users, ...])	Base class to construct data splitters.

time_split

TimeRangeSplitter.

Classes

TimeRangeSplitter(test_size[, n_splits, ...])	Splitter for cross-validation by leave-time-out scheme.

utils

Functions

get_not_seen_mask(train_users, train_items, ...)	Return mask for test interactions that is not in train interactions.

get_not_seen_mask

rectools.model_selection.utils.**get_not_seen_mask**(*train_users: ndarray*, *train_items: ndarray*, *test_users: ndarray*, *test_items: ndarray*) → ndarray

Return mask for test interactions that is not in train interactions.

Parameters

- **train_users** (*np.ndarray*) – Integer array of users in train interactions (it's not a unique users!).
- **train_items** (*np.ndarray*) – Integer array of items in train interactions. Has same length as *train_users*.
- **test_users** (*np.ndarray*) – Integer array of users in test interactions (it's not a unique users!).
- **test_items** (*np.ndarray*) – Integer array of items in test interactions. Has same length as *test_users*.

Returns

Boolean mask of same length as *test_users* (*test_items*). `True` means interaction not present in train.

Return type

np.ndarray

3.8.4 models

Recommendation models (`rectools.models`)

Convenient wrappers for popular recommendation algorithms (ItemKNN, ALS, LightFM), also some custom implementations.

Models

models.DSSMModel models.EASEModel models.ImplicitALSWrapperModel models.ImplicitItemKNNWrapperModel models.LightFMWrapperModel models.PopularModel models.PopularInCategoryModel models.PureSVDModel models.RandomModel

Modules

rectools.models.base	Base model.
rectools.models.dssm	DSSM model.
rectools.models.ease	EASE model.
rectools.models.implicit_als	
rectools.models.implicit_knn	
rectools.models.lightfm	
rectools.models.popular	Popular model.
rectools.models.popular_in_category	Popular in category model.
rectools.models.pure_svd	SVD Model.
rectools.models.random	Random Model.
rectools.models.rank	Implicit ranker model.
rectools.models.utils	Useful functions.
rectools.models.vector	Base classes for vector models.

base

Base model.

Classes

FixedColdRecoModelMixin()	Mixin for models that have fixed cold recommendations.
ModelBase(*args[, verbose])	Base model class.

FixedColdRecoModelMixin

class rectools.models.base.**FixedColdRecoModelMixin**

Bases: object

Mixin for models that have fixed cold recommendations.

Models that use this mixin should implement *_get_cold_reco* method.

Inherited-members

ModelBase

class rectools.models.base.**ModelBase**(**args: Any*, *verbose: int = 0*, ***kwargs: Any*)

Bases: object

Base model class.

Warning: This class should not be used directly. Use derived classes instead.

Inherited-members
Parameters

- **args** (*Any*) –
- **verbose** (*int*) –
- **kwargs** (*Any*) –

Methods

fit(dataset, *args, **kwargs)	Fit model.
recommend(users, dataset, k, filter_viewed)	Recommend items for users.
recommend_to_items(target_items, dataset, k)	Recommend items for target items.

Attributes

recommends_for_cold
recommends_for_warm

fit(*dataset:* Dataset, **args: Any*, ***kwargs: Any*) → T

Fit model.

Parameters

- **dataset** (Dataset) – Dataset with input data.
- **self** (*T*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
self

recommend(*users: Union[Sequence[Hashable], ndarray, Sequence[int]]*, *dataset:* Dataset, *k: int*, *filter_viewed: bool*, *items_to_recommend: Optional[Union[Sequence[Hashable], ndarray, Sequence[int]]] = None*, *add_rank_col: bool = True*, *assume_external_ids: bool = True*) → DataFrame

Recommend items for users.

To use this method model must be fitted.

Parameters

- **users** (*array-like*) – Array of user ids to recommend for. User ids are supposed to be external if *assume_external_ids* is `True` (default). Internal otherwise.
- **dataset** (Dataset) – Dataset with input data. Usually it's the same dataset that was used to fit model.
- **k** (*int*) – Derived number of recommendations for every user. Pay attention that in some cases real number of recommendations may be less than *k*.
- **filter_viewed** (*bool*) – Whether to filter from recommendations items that user has already interacted with. Works only for "hot" users.
- **items_to_recommend** (*array-like, optional, default None*) – Whitelist of item ids. If given, only these items will be used for recommendations. Otherwise all items from dataset will be used. Item ids are supposed to be external if *assume_external_ids* is *True* ` (default). Internal otherwise.
- **add_rank_col** (*bool, default True*) – Whether to add rank column to recommendations. If True column *Columns.Rank* will be added. This column contain integers from 1 to `number of user recommendations`. In any case recommendations are sorted per rank for every user. The lesser the rank the more recommendation is relevant.
- **assume_external_ids** (*bool, default True*) – When `True` all input user and item ids are supposed to be external. Ids in returning recommendations table will be external as well. Internal otherwise. Works faster with `False`.

Returns
Recommendations table with columns *Columns.User*, *Columns.Item*, *Columns.Score`[, `Columns.Rank*]. External user and item ids are used by default. For internal ids set *return_external_ids* to `False`. 1st column contains user ids, 2nd - ids of recommended items sorted by relevance for each user, 3rd - score that model gives for the user-item pair, 4th (present only if *add_rank_col* is `True`) - integers from 1 to number of user recommendations.

Return type
pd.DataFrame

Raises

- **NotFittedError** – If called for not fitted model.
- **TypeError, ValueError** – If arguments have inappropriate type or value
- **ValueError** – If some of given users are warm/cold and model doesn't support such type of users.

recommend_to_items(*target_items: Union[Sequence[Hashable], ndarray, Sequence[int]]*, *dataset:* Dataset, *k: int*, *filter_itself: bool = True*, *items_to_recommend: Optional[Union[Sequence[Hashable], ndarray, Sequence[int]]] = None*, *add_rank_col: bool = True*, *assume_external_ids: bool = True*) → DataFrame

Recommend items for target items.

To use this method model must be fitted.

Parameters

- **target_items** (*array-like*) – Array of item ids to recommend for. Item ids are supposed to be external if *assume_external_ids* is *True* ` (default). Internal otherwise.
- **dataset** (Dataset) – Dataset with input data. Usually it's the same dataset that was used to fit model.
- **k** (*int*) – Derived number of recommendations for every target item. Pay attention that in some cases real number of recommendations may be less than *k*.
- **filter_itself** (*bool, default True*) – If True, item will be excluded from recommendations to itself.
- **items_to_recommend** (*array-like, optional, default None*) – Whitelist of item ids. If given, only these items will be used for recommendations. Otherwise all items from dataset will be used. Item ids are supposed to be external if *assume_external_ids* is *True* ` (default). Internal otherwise.
- **add_rank_col** (*bool, default True*) – Whether to add rank column to recommendations. If True column *Columns.Rank* will be added. This column contain integers from 1 to `number of item recommendations`. In any case recommendations are sorted per rank for every target item. Less rank means more relevant recommendation.
- **assume_external_ids** (*bool, default True*) – When `True` all input item ids are supposed to be external. Ids in returning recommendations table will be external as well. Internal otherwise. Works faster with `False`.

Returns

Recommendations table with columns *Columns.TargetItem*, *Columns.Item*, *Columns.Score `[, `Columns.Rank*]. External item ids are used by default. For internal ids set *return_external_ids* to `False`. 1st column contains target item ids, 2nd - ids of recommended items sorted by relevance for each target item, 3rd - score that model gives for the target-item pair, 4th (present only if *add_rank_col* is `True`) - integers from 1 to number of recommendations.

Return type

pd.DataFrame

Raises

- **NotFittedError** – If called for not fitted model.
- **TypeError, ValueError** – If arguments have inappropriate type or value
- **KeyError** – If some of given target items are not in *dataset.item_id_map*

dssm

DSSM model.

Classes

DSSM(n_factors_user, n_factors_item, ...)	DSSM module for item to item or user to item recommendations.
DSSMModel(train_dataset_type, ...)	Wrapper for *rectools.models.dssm.DSSM*
ItemNet(n_factors, dim_input, activation, ...)	
UserNet(n_factors, dim_input, ...)	

DSSM

class rectools.models.dssm.**DSSM**(*n_factors_user: int, n_factors_item: int, dim_input_user: int, dim_input_item: int, dim_interactions: int, activation: ~typing.Callable[[~torch.Tensor], ~torch.Tensor] = <function elu>, lr: float = 0.01, triplet_loss_margin: float = 0.4, weight_decay: float = 1e-06, log_to_prog_bar: bool = True*)

Bases: LightningModule

DSSM module for item to item or user to item recommendations. This implementation uses triplet loss (see https://en.wikipedia.org/wiki/Triplet_loss) as it's objective function. As an input it expects one-hot encoded item features, one-hot encoded user features and one-hot encoded interactions. Those as easily extracted via *rectools.dataset.Dataset*. During the training cycle item features are propagated through fully connected item network, user features and interactions are propagated through fully connected user network.

Parameters

- **n_factors_user** (*int*) – How many hidden units to use in user network.
- **n_factors_item** (*int*) – How many hidden units to use in item network.
- **dim_input_user** (*int*) – User features dimensionality.
- **dim_input_item** (*int*) – Item features dimensionality.
- **dim_interactions** (*int*) – Interactions dimensionality.
- **activation** (Callable, default *torch.nn.functional.elu*) – Which activation function to use. This function must take a tensor and return a tensor
- **lr** (*float, default 0.01*) – Learning rate.
- **triplet_loss_margin** (*float, default 0.4*) – A nonnegative margin representing the minimum difference between the positive and negative distances required for the loss to be 0. Larger margins penalize cases where the negative examples are not distant enough from the anchors, relative to the positives.
- **weight_decay** (*float, default 1e-6*) – weight decay (L2 penalty).
- **log_to_prog_bar** (*bool, default True*) – Whether to enable logging train and validation losses to progress bar.

Methods

configure_optimizers()	Choose what optimizers and learning-rate schedulers to use in optimization
forward(item_features_pos, ...)	Same as `torch.nn.Module.forward()`.
`inference_items`(dataloader)	
`inference_users`(dataloader)	
training_step(batch, batch_idx)	Compute and return the training loss
validation_step(batch, batch_idx)	Operates on a single batch of data from the validation set.

Attributes

configure_optimizers() → Adam

Choose what optimizers and learning-rate schedulers to use in optimization

Return type
Adam

forward(*item_features_pos: Tensor*, *item_features_neg: Tensor*, *user_features: Tensor*, *interactions: Tensor*) → Tuple[Tensor, Tensor, Tensor]

Same as `torch.nn.Module.forward()`.

Parameters

- ***args** – Whatever you decide to pass into the forward method.
- ****kwargs** – Keyword arguments are also possible.
- **item_features_pos** (*Tensor*) –
- **item_features_neg** (*Tensor*) –
- **user_features** (*Tensor*) –
- **interactions** (*Tensor*) –

Returns
Your model's output

Return type
Tuple[*Tensor*, *Tensor*, *Tensor*]

training_step(*batch: Sequence[Tensor]*, *batch_idx: int*) → Tensor

Compute and return the training loss

Parameters

- **batch** (*Sequence[Tensor]*) –
- **batch_idx** (*int*) –

Return type
Tensor

validation_step(*batch: Sequence[Tensor]*, *batch_idx: int*) → Tensor

Operates on a single batch of data from the validation set. In this step you'd might generate examples or calculate anything of interest like accuracy.

Parameters

- **batch** (*Sequence[Tensor]*) – The output of your data iterable, normally a `DataLoader`.
- **batch_idx** (*int*) – The index of this batch.
- **dataloader_idx** – The index of the dataloader that produced this batch. (only if multiple dataloaders used)

Returns

- `Tensor` - The loss tensor
- `dict` - A dictionary. Can include any keys, but must include the key `'loss'`.
- `None` - Skip to the next batch.

Return type

Tensor

```
# if you have one val dataloader:
def validation_step(self, batch, batch_idx): ...


# if you have multiple val dataloaders:
def validation_step(self, batch, batch_idx, dataloader_idx=0): ...
```

Examples:

```
# CASE 1: A single validation dataset
def validation_step(self, batch, batch_idx):
    x, y = batch

    # implement your own
    out = self(x)
    loss = self.loss(out, y)

    # log 6 example images
    # or generated text... or whatever
    sample_imgs = x[:6]
    grid = torchvision.utils.make_grid(sample_imgs)
    self.logger.experiment.add_image('example_images', grid, 0)

    # calculate acc
    labels_hat = torch.argmax(out, dim=1)
    val_acc = torch.sum(y == labels_hat).item() / (len(y) * 1.0)

    # log the outputs!
    self.log_dict({'val_loss': loss, 'val_acc': val_acc})
```

If you pass in multiple val dataloaders, *validation_step()* will have an additional argument. We recommend setting the default value of 0 so that you can quickly switch between single and multiple dataloaders.

```
# CASE 2: multiple validation dataloaders
def validation_step(self, batch, batch_idx, dataloader_idx=0):
    # dataloader_idx tells you which dataset this is.
    ...
```

Note: If you don't need to validate you don't need to implement this method.

Note: When the *validation_step()* is called, the model has been put in eval mode and PyTorch gradients have been disabled. At the end of validation, the model goes back to training mode and gradients are enabled.

ItemNet

class rectools.models.dssm.**ItemNet**(*n_factors: int, dim_input: int, activation: ~typing.Callable[[~torch.Tensor], ~torch.Tensor] = <function elu>*)

Bases: `Module`

Methods

forward(item_features)	Define the computation performed at every call.

Attributes

Parameters

- **n_factors** (*int*) –
- **dim_input** (*int*) –
- **activation** (*tp.Callable[[torch.Tensor], torch.Tensor]*) –

forward(*item_features: Tensor*) → Tensor

Define the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

Parameters
item_features (*Tensor*) –

Return type
Tensor

UserNet

class rectools.models.dssm.**UserNet**(*n_factors: int, dim_input: int, dim_interactions: int, activation: ~typing.Callable[[~torch.Tensor], ~torch.Tensor] = <function elu>*)

Bases: Module

Methods

forward(user_features, interactions)	Define the computation performed at every call.

Attributes

Parameters

- **n_factors** (*int*) –
- **dim_input** (*int*) –
- **dim_interactions** (*int*) –
- **activation** (*tp.Callable[[torch.Tensor], torch.Tensor]*) –

forward(*user_features: Tensor*, *interactions: Tensor*) → Tensor

Define the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

Parameters

- **user_features** (*Tensor*) –
- **interactions** (*Tensor*) –

Return type
Tensor

ease

EASE model.

Classes

EASEModel([regularization, num_threads, verbose])	Embarrassingly Shallow Autoencoders for Sparse Data model.

implicit_als

Functions

fit_als_with_features_separately_inplace(...)	Fit ALS model with explicit features, explicit features fit separately from latent.
fit_als_with_features_together_inplace(...)	Fit ALS model with explicit features, explicit features fit together with latent.
get_items_vectors(model)	Get items vectors from ALS model as numpy array
get_users_vectors(model)	Get users vectors from ALS model as numpy array

fit_als_with_features_separately_inplace

rectools.models.implicit_als.**fit_als_with_features_separately_inplace**(*model: Union[AlternatingLeastSquares, AlternatingLeastSquares]*, *ui_csr: csr_matrix*, *user_features: Optional[Union[DenseFeatures, SparseFeatures]]*, *item_features: Optional[Union[DenseFeatures, SparseFeatures]]*, *verbose: int = 0*) → None

Fit ALS model with explicit features, explicit features fit separately from latent.

Parameters

- **model** (*AnyAlternatingLeastSquares*) – Base model to fit.
- **ui_csr** (*sparse.csr_matrix*) – Matrix of interactions.
- **user_features** (*(SparseFeatures | DenseFeatures), optional*) – Explicit user features.
- **item_features** (*(SparseFeatures | DenseFeatures), optional*) – Explicit item features.
- **verbose** (*int*) – Whether to print output.

Return type
None

fit_als_with_features_together_inplace

rectools.models.implicit_als.**fit_als_with_features_together_inplace**(*model: Union[AlternatingLeastSquares, AlternatingLeastSquares]*, *ui_csr: csr_matrix*, *user_features: Optional[Union[DenseFeatures, SparseFeatures]]*, *item_features: Optional[Union[DenseFeatures, SparseFeatures]]*, *verbose: int = 0*) → None

Fit ALS model with explicit features, explicit features fit together with latent.

Parameters

- **model** (*AnyAlternatingLeastSquares*) – Base model to fit.
- **ui_csr** (*sparse.csr_matrix*) – Matrix of interactions.
- **user_features** (*(SparseFeatures | DenseFeatures), optional*) – Explicit user features.
- **item_features** (*(SparseFeatures | DenseFeatures), optional*) – Explicit item features.
- **verbose** (*int*) – Whether to print output.

Return type
None

get_items_vectors

rectools.models.implicit_als.**get_items_vectors**(*model: Union[AlternatingLeastSquares, AlternatingLeastSquares]*) → ndarray

Get items vectors from ALS model as numpy array

Parameters
model (*AnyAlternatingLeastSquares*) – Model to get vectors from. Can be CPU or GPU model

Returns
Item vectors

Return type
np.ndarray

get_users_vectors

rectools.models.implicit_als.**get_users_vectors**(*model: Union[AlternatingLeastSquares, AlternatingLeastSquares]*) → ndarray

Get users vectors from ALS model as numpy array

Parameters
model (*AnyAlternatingLeastSquares*) – Model to get vectors from. Can be CPU or GPU model

Returns
User vectors

Return type
np.ndarray

Classes

ImplicitALSWrapperModel(model[, verbose, ...])	Wrapper for *implicit.als.AlternatingLeastSquares* with possibility to use explicit features and GPU support.

implicit_knn

Classes

ImplicitItemKNNWrapperModel(model[, verbose])	Wrapper for *implicit.nearest_neighbours.ItemItemRecommender* and its successors.

lightfm

Classes

LightFMWrapperModel(model[, epochs, ...])	Wrapper for *lightfm.LightFM*.

popular

Popular model.

Classes

PopularModel([popularity, period, ...])	Model generating recommendations based on popularity of items.
Popularity(value)	Types of popularity

Popularity

class rectools.models.popular.**Popularity**(*value*)

Bases: Enum

Types of popularity

Inherited-members

Attributes

N_USERS
N_INTERACTIONS
MEAN_WEIGHT
SUM_WEIGHT

popular_in_category

Popular in category model.

Classes

MixingStrategy(value)	Types of mixing strategy
PopularInCategoryModel(category_feature[, ...])	Model generating recommendations based on popularity of items.
RatioStrategy(value)	Types of ratio strategy

MixingStrategy

class rectools.models.popular_in_category.**MixingStrategy**(*value*)

Bases: Enum

Types of mixing strategy

Inherited-members

Attributes

ROTATE
GROUP

RatioStrategy

class rectools.models.popular_in_category.**RatioStrategy**(*value*)

Bases: Enum

Types of ratio strategy

Inherited-members

Attributes

EQUAL
PROPORTIONAL

pure_svd

SVD Model.

Classes

PureSVDModel([factors, verbose])	PureSVD matrix factorization model.

random

Random Model.

Classes

RandomModel([random_state, verbose])	Model generating random recommendations.
_RandomGen([random_state])	
_RandomSampler(values, random_gen)	

_RandomGen

class rectools.models.random._RandomGen(*random_state: Optional[int] = None*)

Bases: object

Inherited-members

Parameters

random_state (*Optional[int]*) –

_RandomSampler

class rectools.models.random._RandomSampler(*values: ndarray*, *random_gen: _RandomGen*)

Bases: object

Inherited-members

Parameters

- **values** (*ndarray*) –
- **random_gen** (_RandomGen) –

Methods

sample(n)

rank

Implicit ranker model.

Classes

Distance(value)	Distance metric
ImplicitRanker(distance, subjects_factors, ...)	Ranker model which uses implicit library matrix factorization topk method.

Distance

class rectools.models.rank.Distance(*value*)

Bases: Enum

Distance metric

Inherited-members

Attributes

DOT
COSINE
EUCLIDEAN

ImplicitRanker

class rectools.models.rank.**ImplicitRanker**(*distance: Distance*, *subjects_factors: Union[ndarray, csr_matrix]*, *objects_factors: ndarray*)

Bases: `object`

Ranker model which uses implicit library matrix factorization topk method.

This ranker is suitable for the following cases of scores calculation: 1. subject_embeddings.dot(objects_embeddings) 2. subject_interactions.dot(item-item-similarities)

Parameters

- **distance** (*Distance*) – Distance metric.
- **subjects_factors** (*np.ndarray | sparse.csr_matrix*) – Array of subjects embeddings, shape (n_subjects, n_factors). For item-item similarity models subjects vectors from ui_csr are viewed as factors.
- **objects_factors** (*np.ndarray*) – Array with embeddings of all objects, shape (n_objects, n_factors). For item-item similarity models item similarity vectors are viewed as factors.

Inherited-members

Methods

rank(subject_ids, k[, filter_pairs_csr, ...])	Rank objects to proceed inference using implicit library topk cpu method.

rank(*subject_ids: Union[Sequence[int], ndarray]*, *k: int*, *filter_pairs_csr: Optional[csr_matrix] = None*, *sorted_object_whitelist: Optional[ndarray] = None*, *num_threads: int = 0*) → Tuple[Union[Sequence[int], ndarray], Union[Sequence[int], ndarray], Union[Sequence[float], ndarray]]

Rank objects to proceed inference using implicit library topk cpu method.

Parameters

- **subject_ids** (*csr_matrix*) – Array of ids to recommend for.
- **k** (*int*) – Derived number of recommendations for every subject id.
- **filter_pairs_csr** (sparse.csr_matrix, optional, default `None`) – Subject-object interactions that should be filtered from recommendations. This is relevant for u2i case.
- **sorted_object_whitelist** (sparse.csr_matrix, optional, default `None`) – Whitelist of object ids. If given, only these items will be used for recommendations. Otherwise all items from dataset will be used.

- **num_threads** (*int, default 0*) – Will be used as *num_threads* parameter for *implicit.cpu.topk.topk*.

Returns
Array of subject ids, array of recommended items, sorted by score descending and array of scores.

Return type
(InternalIds, InternalIds, Scores)

utils

Useful functions.

Functions

get_viewed_item_ids(user_items, user_id)	Return indices of items that user has interacted with.
recommend_from_scores(scores, k[, ...])	Prepare top-k recommendations for a user.

get_viewed_item_ids

rectools.models.utils.**get_viewed_item_ids**(*user_items: csr_matrix*, *user_id: int*) → ndarray

Return indices of items that user has interacted with.

Parameters

- **user_items** (*csr_matrix*) – Matrix of interactions.
- **user_id** (*int*) – Internal user id.

Returns
Internal item indices that user has interacted with.

Return type
np.ndarray

recommend_from_scores

rectools.models.utils.**recommend_from_scores**(*scores: ndarray*, *k: int*, *sorted_blacklist: Optional[ndarray] = None*, *sorted_whitelist: Optional[ndarray] = None*, *ascending: bool = False*) → Tuple[ndarray, ndarray]

Prepare top-k recommendations for a user.

Recommendations are sorted by item scores for this particular user. Recommendations can be filtered according to whitelist and blacklist.

If *I* - set of all items, *B* - set of blacklist items, *W* - set of whitelist items, then:

- if *W* is `None`, then for recommendations will be used *I* - *B* set of items
- if *W* is not `None`, then for recommendations will be used *W* - *B* set of items

Parameters

- **scores** (*np.ndarray*) – Array of floats. Scores of relevance of all items for this user. Shape (n_items,).
- **k** (*int*) – Desired number of final recommendations. If, after applying white- and black-list, number of available items *n_available* is less than *k*, then *n_available* items will be returned without warning.
- **sorted_blacklist** (np.ndarray, optional, default None) – Array of unique ints. Sorted inner item ids to exclude from recommendations.
- **sorted_whitelist** (np.ndarray, optional, default None) – Array of unique ints. Sorted inner item ids to use in recommendations.
- **ascending** (*bool, default False*) – If False, sorting by descending of score, use when score are metric of similarity. If True, sorting by ascending of score, use when score are distance.

Returns
Array of recommended items, sorted by score descending.

Return type
np.ndarray

vector

Base classes for vector models.

Classes

Factors(embeddings[, biases])	Embeddings and biases
VectorModel(*args[, verbose])	Base class for models that represents users and items as vectors

Factors

class rectools.models.vector.**Factors**(*embeddings: ndarray*, *biases: Optional[ndarray] = None*)

Bases: object

Embeddings and biases

Inherited-members
Parameters

- **embeddings** (*ndarray*) –
- **biases** (*Optional[ndarray]*) –

Attributes

embeddings
biases

VectorModel

class rectools.models.vector.**VectorModel**(**args: Any*, *verbose: int = 0*, ***kwargs: Any*)

Bases: *ModelBase*

Base class for models that represents users and items as vectors

Inherited-members

Parameters

- **args** (*Any*) –
- **verbose** (*int*) –
- **kwargs** (*Any*) –

Methods

fit(dataset, *args, **kwargs)	Fit model.
recommend(users, dataset, k, filter_viewed)	Recommend items for users.
recommend_to_items(target_items, dataset, k)	Recommend items for target items.

Attributes

i2i_dist
n_threads
recommends_for_cold
recommends_for_warm
u2i_dist

3.8.5 tools

Tools (rectools.tools)

Various useful instruments to make recommendations better.

Tools

tools.ItemToItemAnnRecommender tools.UserToItemAnnRecommender

Modules

rectools.tools.ann	Approximate Nearest Neighbours accelerators

ann

Approximate Nearest Neighbours accelerators

Classes

BaseNmslibRecommender(item_vectors, item_id_map)	Class implements base constructor parameters, pickling protocol and sort-truncate logic for *UserToItemAnnRecommender* and *ItemToItemAnnRecommender*.
ItemToItemAnnRecommender(item_vectors, ...)	Class implements item-to-item ANN recommender.
UserToItemAnnRecommender(user_vectors, ...)	Class implements user to item ANN recommender.

BaseNmslibRecommender

class rectools.tools.ann.**BaseNmslibRecommender**(*item_vectors: np.ndarray*, *item_id_map: tp.Union[*IdMap*, tp.Dict[ExternalId, InternalId]]*, *index_top_k: int = 0*, *index_init_params: tp.Optional[tp.Dict[str, str]] = None*, *index_query_time_params: tp.Optional[tp.Dict[str, int]] = None*, *create_index_params: tp.Optional[tp.Dict[str, int]] = None*, *index: tp.Optional[nmslib.FloatIndex] = None*)

Bases: object

Class implements base constructor parameters, pickling protocol and sort-truncate logic for *UserToItemAnnRecommender* and *ItemToItemAnnRecommender*.

Parameters

- **item_vectors** (*ndarray*) – Ndarray of item latent features of size `(N, K)`, where *N* is the number if items and *K* is the number of features.
- **item_id_map** (*dict(hashable, int) | rectools.datasets.IdMap*) – Mappings from external item ids to internal item ids used by recommender. Values must be positive integers.

- **index_top_k** (*int, default 0*) – Number of items to return per knn query (in addition to *top_n* passed to *get_item_list_for_user*, *get_item_list_for_user_batch*, *get_item_list_for_item* or *get_item_list_for_item_batch*). In this case nmslib index query. This might be important in order to account for filters. See *self.index.knnQueryBatch* in `_compute_sorted_similar'
- **index_init_params** (*optional(dict(str, str)), default None*) – NMSLIB initialization parameters. See nmslib documentation. In case of None defaults to reasonable parameters.
- **index_query_time_params** (*optional(dict(str, str)), default None*) – NMSLIB query time parameters. See nmslib documentation. In case of None defaults to reasonable parameters.
- **create_index_params** (*optional(dict(str, str)), default None*) – NMSLIB index creation parameters. See nmslib documentation. In case of None defaults to reasonable parameters.
- **index** (*FloatIndex, optional*) – Optional instance of FloatIndex. Exists for outside initialization.

See also:

UserToItemAnnRecommender, *ItemToItemAnnRecommender*

Inherited-members
Parameters

- **item_vectors** (*np.ndarray*) –
- **item_id_map** (*tp.Union[IdMap, tp.Dict[ExternalId, InternalId]]*) –
- **index_top_k** (*int*) –
- **index_init_params** (*tp.Optional[tp.Dict[str, str]]*) –
- **index_query_time_params** (*tp.Optional[tp.Dict[str, int]]*) –
- **create_index_params** (*tp.Optional[tp.Dict[str, int]]*) –
- **index** (*tp.Optional[nmslib.FloatIndex]*) –

Methods

fit([verbose])	Create and fit *nmslib* index.

fit(*verbose: bool = False*) → T

Create and fit *nmslib* index.

Parameters

- **verbose** (*bool*) – Verbosity switch, see *NMSLIB* documentation.
- **self** (*T*) –

Returns
Returns self.

Return type
BaseNmslibRecommender

3.8.6 visuals

Visualization tools (rectools.visuals)

Instruments to visualize recommender models performance

Recommendations visualization

visuals.VisualApp - Jupyter app for visual comparison of recommendations *visuals.ItemToItemVisualApp* - Jupyter app for visual comparison of item-to-item recommendations

Modules

rectools.visuals.visual_app

visual_app

Classes

AppDataStorage(is_u2i, id_col, ...)	Storage and processing of data for *VisualApp* widgets.
ItemToItemVisualApp(data_storage[, ...])	Jupyter widgets app for item-to-item recommendations visualization and models comparison.
StorageFiles()	Fixed file names for *AppDataStorage* saving and loading.
VisualApp(data_storage[, auto_display, ...])	Jupyter widgets app for user-to-item recommendations visualization and models comparison.
VisualAppBase(data_storage[, auto_display, ...])	Jupyter widgets app for recommendations visualization and models comparison.

AppDataStorage

class rectools.visuals.visual_app.**AppDataStorage**(*is_u2i: bool*, *id_col: str*, *selected_requests: Dict[Hashable, Hashable]*, *grouped_interactions: Dict[Hashable, DataFrame]*, *grouped_reco: Dict[Hashable, Dict[Hashable, DataFrame]]*)

Bases: object

Storage and processing of data for *VisualApp* widgets. This class is not meant to be used directly. Use *VisualApp* or *ItemToItemVisualApp* class instead

Inherited-members
Parameters

- **is_u2i** (*bool*) –
- **id_col** (*str*) –
- **selected_requests** (*Dict[Hashable, Hashable]*) –
- **grouped_interactions** (*Dict[Hashable, DataFrame]*) –

- **grouped_reco** (*Dict[Hashable, Dict[Hashable, DataFrame]]*) –

Methods

from_raw(reco, item_data[, ...])	Create data storage for VisualApp from raw data.
load(folder_name)	Load prepared data for VisualApp widgets.
save(folder_name[, overwrite])	Save stored data for *VisualApp* widgets.

Attributes

is_u2i
id_col
selected_requests
grouped_interactions
grouped_reco

model_names	Names of recommendation models for comparison
request_names	Names of selected requests for comparison

classmethod from_raw(*reco: Union[DataFrame, Dict[Hashable, DataFrame]]*, *item_data: DataFrame*, *selected_requests: Optional[Dict[Hashable, Hashable]] = None*, *is_u2i: bool = True*, *n_random_requests: int = 0*, *interactions: Optional[DataFrame] = None*) → *AppDataStorage*

Create data storage for VisualApp from raw data. This class is not meant to be used directly. Use *VisualApp* or *ItemToItemVisualApp* class instead.

Parameters

- **reco** (*tp.Union[pd.DataFrame, TablesDict]*) – Recommendations from different models in a form of a pd.DataFrame or a dict. In DataFrame form model names must be specified in *Columns.Model* column. In dict form model names are supposed to be dict keys.
- **item_data** (*pd.DataFrame*) – Data for items that is used for visualisation in both interactions and recommendations widgets.
- **selected_requests** (tp.Optional[tp.Dict[tp.Hashable, ExternalId]], default `None`) – Predefined requests (users or items) that will be displayed in widgets. Request names must be specified as keys of the dict and ids as values of the dict.
- **is_u2i** (bool, default `True`) – Is this a user-to-item recommendation case (opposite to item-to-item).
- **n_random_requests** (*int, default 0*) – Number of random requests to add for visualization from targets in recommendation tables.
- **interactions** (tp.Optional[pd.DataFrame], default `None`) – Table with interactions history for users. Only needed for u2i case.

Returns
Data storage class for visualisation widgets.

Return type
AppDataStorage

classmethod load(*folder_name: str*) → *AppDataStorage*

Load prepared data for VisualApp widgets. This method is not meant to be used directly. Use *VisualApp* or *ItemToItemVisualApp* class methods instead.

Parameters
folder_name (*str*) – Folder where data was saved earlier.

Returns
Data storage class for visualisation widgets.

Return type
AppDataStorage

property model_names: List[Hashable]

Names of recommendation models for comparison

property request_names: List[Hashable]

Names of selected requests for comparison

save(*folder_name: str*, *overwrite: bool = False*) → None

Save stored data for *VisualApp* widgets. This method is not meant to be used directly. Use *VisualApp* or *ItemToItemVisualApp* class methods instead.

Parameters

- **folder_name** (*str*) – Destination folder for data.
- **overwrite** (bool, default `False`) – Allow to overwrite in the folder files if they already exist.

Return type
None

StorageFiles

class rectools.visuals.visual_app.**StorageFiles**

Bases: `object`

Fixed file names for *AppDataStorage* saving and loading.

Inherited-members

Attributes

`Interactions`
`Recommendations`
`Requests`

VisualAppBase

class rectools.visuals.visual_app.**VisualAppBase**(*data_storage: AppDataStorage*, *auto_display: bool = True*, *formatters: Optional[Dict[str, Callable]] = None*, *rows_limit: int = 20*, *min_width: int = 50*)

Bases: `object`

Jupyter widgets app for recommendations visualization and models comparison. Warning! This is a base class. Do not create instances of this class directly. Use derived classes *construct* methods instead.

Inherited-members

Parameters

- **data_storage** (*AppDataStorage*) –
- **auto_display** (*bool*) –
- **formatters** (*Optional[Dict[str, Callable]]*) –
- **rows_limit** (*int*) –
- **min_width** (*int*) –

Methods

display()	Display full VisualApp widget
load(folder_name[, auto_display, ...])	Create widgets from data that was processed and saved earlier.
save(folder_name[, overwrite])	Save stored data to re-create widgets when necessary.

display() → None

Display full VisualApp widget

Return type
None

classmethod **load**(*folder_name: str*, *auto_display: bool = True*, *formatters: Optional[Dict[str, Callable]] = None*, *rows_limit: int = 20*, *min_width: int = 100*) → VisualAppT

Create widgets from data that was processed and saved earlier.

Parameters

- **folder_name** (*str*) – Destination folder for data.
- **auto_display** (bool, optional, default `True`) – Display widgets right after initialization.
- **formatters** (tp.Optional[tp.Dict[str, tp.Callable]], optional, default `None`) – Formatter functions to apply to columns elements in the sections of interactions and recommendations. Keys of the dict must be columns names (item_data, interactions and recommendations columns can be specified here). Values bust be functions that will be applied to corresponding columns elements. The result of each function must be a unicode string that represents html code. Formatters can be used to format text, create links and display images with html.
- **rows_limit** (*int, optional, default 20*) – Maximum number of rows to display in the sections of interactions and recommendations.

- **min_width** (*int, optional, default 100*) – Minimum column width in pixels for dataframe columns in widgets output. Must be greater then 10.

Returns
Jupyter widgets for recommendations visualization.

Return type
VisualAppBase

save(*folder_name: str*, *overwrite: bool = False*) → None

Save stored data to re-create widgets when necessary. Use *VisualAppBase.load* class method for re-creation or any other child classes (*VisualApp*, *ItemToItemVisualApp*).

Parameters

- **folder_name** (*str*) – Destination folder for data.
- **overwrite** (bool, default `False`) – Allow to overwrite in the folder files if they already exist.

Return type
None

3.9 Tutorials

See tutorials here: https://github.com/MobileTeleSystems/RecTools/tree/main/examples

3.9.1 Simple example of building recommendations with RecTools

- Building simple model
- Visual recommendations checking

[ ]:
```
import numpy as np
import pandas as pd

from implicit.nearest_neighbours import TFIDFRecommender

from rectools import Columns
from rectools.dataset import Dataset
from rectools.models import ImplicitItemKNNWrapperModel
```

Load data

[2]:
```
%%time
!wget -q https://files.grouplens.org/datasets/movielens/ml-1m.zip -O ml-1m.zip
!unzip -o ml-1m.zip
!rm ml-1m.zip
```

```
Archive:  ml-1m.zip
  inflating: ml-1m/movies.dat
  inflating: ml-1m/ratings.dat
  inflating: ml-1m/README
  inflating: ml-1m/users.dat
```

(continues on next page)

(continued from previous page)

```
CPU times: user 134 ms, sys: 415 ms, total: 548 ms
Wall time: 4.39 s
```

[2]:
```
%%time
ratings = pd.read_csv(
    "ml-1m/ratings.dat",
    sep="::",
    engine="python",  # Because of 2-chars separators
    header=None,
    names=[Columns.User, Columns.Item, Columns.Weight, Columns.Datetime],
)
print(ratings.shape)
ratings.head()
```

```
(1000209, 4)
CPU times: user 5.76 s, sys: 409 ms, total: 6.17 s
Wall time: 6.16 s
```

[2]:

	user_id	item_id	weight	datetime
0	1	1193	5	978300760
1	1	661	3	978302109
2	1	914	3	978301968
3	1	3408	4	978300275
4	1	2355	5	978824291

[3]:
```
%%time
movies = pd.read_csv(
    "ml-1m/movies.dat",
    sep="::",
    engine="python",  # Because of 2-chars separators
    header=None,
    names=[Columns.Item, "title", "genres"],
    encoding_errors="ignore",
)
print(movies.shape)
movies.head()
```

```
(3883, 3)
CPU times: user 9.55 ms, sys: 1.62 ms, total: 11.2 ms
Wall time: 10.4 ms
```

[3]:

	item_id	title	genres
0	1	Toy Story (1995)	Animation|Children's|Comedy
1	2	Jumanji (1995)	Adventure|Children's|Fantasy
2	3	Grumpier Old Men (1995)	Comedy|Romance
3	4	Waiting to Exhale (1995)	Comedy|Drama
4	5	Father of the Bride Part II (1995)	Comedy

Build model

[4]:
```
# Prepare a dataset to build a model
dataset = Dataset.construct(ratings)
```

[5]:
```
%%time
# Fit model and generate recommendations for all users
model = ImplicitItemKNNWrapperModel(TFIDFRecommender(K=10))
model.fit(dataset)
recos = model.recommend(
    users=ratings[Columns.User].unique(),
    dataset=dataset,
    k=10,
    filter_viewed=True,
)
```

```
CPU times: user 6.05 s, sys: 274 ms, total: 6.32 s
Wall time: 1.42 s
```

[6]:
```
# Sample of recommendations - it's sorted by relevance (= rank) for each user
recos.head()
```

[6]:

	user_id	item_id	score	rank
0	1	364	20.436578	1
1	1	1196	15.716834	2
2	1	318	15.625371	3
3	1	2096	14.876911	4
4	1	2571	12.718620	5

Check recommendations

[7]:
```
# Select random user, see history of views and reco for this user
user_id = 3883
user_viewed = ratings.query("user_id == @user_id").merge(movies, on="item_id")
user_recos = recos.query("user_id == @user_id").merge(movies, on="item_id")
```

[8]:
```
# History, but only films that user likes
user_viewed.query("weight > 3")
```

[8]:

	user_id	item_id	weight	datetime	title	genres
0	3883	2997	5	967134212	Being John Malkovich (1999)	Comedy
2	3883	1265	5	967134285	Groundhog Day (1993)	Comedy|Romance
4	3883	2858	5	965822230	American Beauty (1999)	
10	3883	2369	4	965822136	Desperately Seeking Susan (1985)	
14	3883	3189	4	965822296	My Dog Skip (1999)	
16	3883	1784	4	965822136	As Good As It Gets (1997)	
17	3883	2599	4	967134250	Election (1999)	
18	3883	34	4	967134285	Babe (1995)	

(continues on next page)

(continued from previous page)

```
4               Comedy|Drama
10             Comedy|Romance
14                     Comedy
16              Comedy|Drama
17                     Comedy
18  Children's|Comedy|Drama
```

[9]:
```
# Recommendations
user_recos.sort_values("rank")
```

[9]:
```
   user_id  item_id      score  rank                              title  \
0     3883     2396  13.991358     1         Shakespeare in Love (1998)
1     3883     2762  10.249648     2            Sixth Sense, The (1999)
2     3883      318   7.728188     3  Shawshank Redemption, The (1994)
3     3883      608   7.617913     4                      Fargo (1996)
4     3883      356   5.674010     5                Forrest Gump (1994)
5     3883     2395   5.508895     6                    Rushmore (1998)
6     3883      223   5.398012     7                     Clerks (1994)
7     3883      593   5.335058     8  Silence of the Lambs, The (1991)
8     3883      296   4.828189     9                Pulp Fiction (1994)
9     3883     2959   4.615653    10                  Fight Club (1999)

                genres
0        Comedy|Romance
1              Thriller
2                 Drama
3  Crime|Drama|Thriller
4    Comedy|Romance|War
5                Comedy
6                Comedy
7        Drama|Thriller
8           Crime|Drama
9                 Drama
```

Here is the simple example, we only used ratings to train the model and we only prepared recommendations for users who have rated movies before. But some models allow you to use explicit features, e.g. user age or item genre. And some models allow you to generate recommendations for users that have not rated any movies before. See documentation for the details.

[ ]:
```
```

3.9.2 Example of model selection using cross-validation with RecTools

- CV split
- Training a variety of models
- Measuring a variety of metrics

[1]:
```
from pprint import pprint

import numpy as np
```

(continues on next page)

(continued from previous page)

```
import pandas as pd

from tqdm.auto import tqdm

from implicit.nearest_neighbours import TFIDFRecommender, BM25Recommender
from implicit.als import AlternatingLeastSquares

from rectools import Columns
from rectools.dataset import Dataset
from rectools.metrics import Precision, Recall, MeanInvUserFreq, Serendipity, calc_
↪metrics
from rectools.models import ImplicitItemKNNWrapperModel, RandomModel, PopularModel
from rectools.model_selection import TimeRangeSplitter, cross_validate
```

Load data

[2]:
```
%%time
!wget -q https://files.grouplens.org/datasets/movielens/ml-1m.zip -O ml-1m.zip
!unzip -o ml-1m.zip
!rm ml-1m.zip
```

```
Archive:  ml-1m.zip
  inflating: ml-1m/movies.dat
  inflating: ml-1m/ratings.dat
  inflating: ml-1m/README
  inflating: ml-1m/users.dat
CPU times: user 53 ms, sys: 40.2 ms, total: 93.2 ms
Wall time: 3.15 s
```

[3]:
```
%%time
ratings = pd.read_csv(
    "ml-1m/ratings.dat",
    sep="::",
    engine="python",  # Because of 2-chars separators
    header=None,
    names=[Columns.User, Columns.Item, Columns.Weight, Columns.Datetime],
)
print(ratings.shape)
ratings.head()
```

```
(1000209, 4)
CPU times: user 4.51 s, sys: 198 ms, total: 4.71 s
Wall time: 4.76 s
```

[3]:

	user_id	item_id	weight	datetime
0	1	1193	5	978300760
1	1	661	3	978302109
2	1	914	3	978301968
3	1	3408	4	978300275
4	1	2355	5	978824291

[4]:
```
ratings["user_id"].nunique(), ratings["item_id"].nunique()
```

[4]:
```
(6040, 3706)
```

[5]:
```
ratings["weight"].value_counts()
```

[5]:
```
4    348971
3    261197
5    226310
2    107557
1     56174
Name: weight, dtype: int64
```

[6]:
```
ratings["datetime"] = pd.to_datetime(ratings["datetime"] * 10 ** 9)
print("Time period")
ratings["datetime"].min(), ratings["datetime"].max()
```

```
Time period
```

[6]:
```
(Timestamp('2000-04-25 23:05:32'), Timestamp('2003-02-28 17:49:50'))
```

Create `Dataset` class. It's a wrapper for interactions. User and item features can also be added (see next examples for details).

[7]:
```
%%time
dataset = Dataset.construct(ratings)
```

```
CPU times: user 54.7 ms, sys: 15.5 ms, total: 70.3 ms
Wall time: 70.1 ms
```

Prepare cross-validation splitter

We'll use last 3 periods of 2 weeks to validate our models.

[8]:
```
n_splits = 3

splitter = TimeRangeSplitter(
    test_size="14D",
    n_splits=n_splits,
    filter_already_seen=True,
    filter_cold_items=True,
    filter_cold_users=True,
)
```

[9]:
```
splitter.get_test_fold_borders(dataset.interactions)
```

[9]:
```
[(Timestamp('2003-01-18 00:00:00', freq='14D'),
  Timestamp('2003-02-01 00:00:00', freq='14D')),
 (Timestamp('2003-02-01 00:00:00', freq='14D'),
  Timestamp('2003-02-15 00:00:00', freq='14D')),
 (Timestamp('2003-02-15 00:00:00', freq='14D'),
  Timestamp('2003-03-01 00:00:00', freq='14D'))]
```

For test folds left border is always included in fold and the right one is excluded.

Train folds don't have left border, and the right one is always excluded.

Train models

[10]:
```
# Take few simple models to compare
models = {
    "random": RandomModel(random_state=42),
    "popular": PopularModel(),
    "most_raited": PopularModel(popularity="sum_weight"),
    "tfidf_k=5": ImplicitItemKNNWrapperModel(model=TFIDFRecommender(K=5)),
    "tfidf_k=10": ImplicitItemKNNWrapperModel(model=TFIDFRecommender(K=10)),
    "bm25_k=10_k1=0.05_b=0.1": ImplicitItemKNNWrapperModel(model=BM25Recommender(K=5,␣
↪K1=0.05, B=0.1)),
}

# We will calculate several classic (precision@k and recall@k) and "beyond accuracy"␣
↪metrics
metrics = {
    "prec@1": Precision(k=1),
    "prec@10": Precision(k=10),
    "recall": Recall(k=10),
    "novelty": MeanInvUserFreq(k=10),
    "serendipity": Serendipity(k=10),
}

K_RECS = 10
```

[11]:
```
%%time

# For each fold generate train and test part of dataset
# Then fit every model, generate recommendations and calculate metrics

cv_results = cross_validate(
    dataset=dataset,
    splitter=splitter,
    models=models,
    metrics=metrics,
    k=K_RECS,
    filter_viewed=True,
)
```

```
CPU times: user 14.2 s, sys: 714 ms, total: 14.9 s
Wall time: 14.9 s
```

We can get some split stats

[12]:
```
pd.DataFrame(cv_results["splits"])
```

[12]:

	i_split	start	end	train	train_users	train_items	test	\
0	0	2003-01-18	2003-02-01	998083	6040	3706	630	
1	1	2003-02-01	2003-02-15	998713	6040	3706	899	
2	2	2003-02-15	2003-03-01	999612	6040	3706	597	

	test_users	test_items
0	75	540
1	57	704

(continues on next page)

(continued from previous page)

```
2          66         501
```

And the main result is metrics

[13]:
```
pd.DataFrame(cv_results["metrics"])
```

[13]:

	model	i_split	prec@1	prec@10	recall	novelty	serendipity
0	random	0	0.000000	0.000000	0.000000	6.539622	0.000000
1	popular	0	0.053333	0.024000	0.037410	1.580736	0.000123
2	most_raited	0	0.053333	0.026667	0.042251	1.592543	0.000151
3	tfidf_k=5	0	0.053333	0.021333	0.023866	2.361189	0.000465
4	tfidf_k=10	0	0.026667	0.021333	0.039926	2.137451	0.000327
5	bm25_k=10_k1=0.05_b=0.1	0	0.026667	0.029333	0.046645	1.781881	0.000271
6	random	1	0.000000	0.001754	0.017544	6.489885	0.000054
7	popular	1	0.052632	0.057895	0.015707	1.588414	0.000183
8	most_raited	1	0.035088	0.056140	0.009919	1.600628	0.000155
9	tfidf_k=5	1	0.052632	0.057895	0.048591	2.326116	0.002616
10	tfidf_k=10	1	0.052632	0.052632	0.010033	2.143504	0.000917
11	bm25_k=10_k1=0.05_b=0.1	1	0.070175	0.059649	0.010321	1.809416	0.000341
12	random	2	0.000000	0.004545	0.000956	6.535055	0.000608
13	popular	2	0.045455	0.042424	0.039984	1.656638	0.000450
14	most_raited	2	0.045455	0.039394	0.024521	1.668108	0.000332
15	tfidf_k=5	2	0.090909	0.050000	0.039606	2.378988	0.001500
16	tfidf_k=10	2	0.060606	0.051515	0.053531	2.206921	0.001346
17	bm25_k=10_k1=0.05_b=0.1	2	0.090909	0.039394	0.038426	1.901316	0.000397

Let's now aggregate metrics by folds and compare models

[14]:
```
pivot_results = (
    pd.DataFrame(cv_results["metrics"])
    .drop(columns="i_split")
    .groupby(["model"], sort=False)
```

(continues on next page)

(continued from previous page)

```
    .agg(["mean", "std"])
)
mean_metric_subset = [(metric, "mean") for metric in pivot_results.columns.levels[0]]
(
    pivot_results.style
    .highlight_min(subset=mean_metric_subset, color='lightcoral', axis=0)
    .highlight_max(subset=mean_metric_subset, color='lightgreen', axis=0)
)
```

[14]: <pandas.io.formats.style.Styler at 0x7fee689b8c70>

	prec@1		prec@10		recall		novelty		serendipity	
	mean	std	mean	std	mean	std	mean	std	mean	std
model										
random	0.000000	0.000000	0.002100	0.002292	0.006167	0.009865	6.521521	0.027493	0.000220	0.000336
popular	0.050473	0.004360	0.041440	0.016969	0.031034	0.013335	1.608596	0.041782	0.000252	0.000174
most_raited	0.044625	0.009151	0.040734	0.014782	0.025564	0.016192	1.620426	0.041491	0.000213	0.000103
tfidf_k=5	0.065625	0.021900	0.043076	0.019239	0.037354	0.012516	2.355431	0.026902	0.001527	0.001076
tfidf_k=10	0.046635	0.017747	0.041827	0.017757	0.034497	0.022252	2.162626	0.038480	0.000863	0.000512
bm25_k=10_k1=0.05_b=0.1	0.062584	0.032787	0.042792	0.015441	0.031797	0.019048	1.830871	0.062541	0.000337	0.000063

3.9.3 Examples of calculating different metrics with RecTools

- Initializing different metrics
- Calculating a value of a single metric
- Calculating metric values per user
- Calculating values of a bunch of metrics using only one function

[2]:
```
import numpy as np
import pandas as pd

from implicit.nearest_neighbours import TFIDFRecommender

from rectools import Columns
from rectools.dataset import Dataset
from rectools.metrics import (
    Precision,
    Accuracy,
    NDCG,
    IntraListDiversity,
    Serendipity,
    calc_metrics,
)
from rectools.metrics.distances import PairwiseHammingDistanceCalculator
from rectools.models import ImplicitItemKNNWrapperModel
```

Load data

[3]:
```
%%time
!wget -q https://files.grouplens.org/datasets/movielens/ml-1m.zip -O ml-1m.zip
!unzip -o ml-1m.zip
!rm ml-1m.zip
```

```
Archive:  ml-1m.zip
  inflating: ml-1m/movies.dat
  inflating: ml-1m/ratings.dat
  inflating: ml-1m/README
  inflating: ml-1m/users.dat
CPU times: user 39.5 ms, sys: 44.8 ms, total: 84.3 ms
Wall time: 3.22 s
```

[4]:
```
%%time
ratings = pd.read_csv(
    "ml-1m/ratings.dat",
    sep="::",
    engine="python",  # Because of 2-chars separators
    header=None,
    names=[Columns.User, Columns.Item, Columns.Weight, Columns.Datetime],
)
print(ratings.shape)
ratings.head()
```

```
(1000209, 4)
CPU times: user 3.51 s, sys: 270 ms, total: 3.78 s
Wall time: 3.77 s
```

[4]:

	user_id	item_id	weight	datetime
0	1	1193	5	978300760
1	1	661	3	978302109
2	1	914	3	978301968
3	1	3408	4	978300275
4	1	2355	5	978824291

[5]:
```
ratings["datetime"] = pd.to_datetime(ratings["datetime"] * 10 ** 9)
ratings["datetime"].min(), ratings["datetime"].max()
```

[5]:
```
(Timestamp('2000-04-25 23:05:32'), Timestamp('2003-02-28 17:49:50'))
```

[6]:
```
%%time
movies = pd.read_csv(
    "ml-1m/movies.dat",
    sep="::",
    engine="python",  # Because of 2-chars separators
    header=None,
    names=[Columns.Item, "title", "genres"],
    encoding_errors="ignore",
)
print(movies.shape)
movies.head()
```

```
(3883, 3)
CPU times: user 9.53 ms, sys: 518 µs, total: 10 ms
Wall time: 9.36 ms
```

[6]:

	item_id	title	genres
0	1	Toy Story (1995)	Animation|Children's|Comedy
1	2	Jumanji (1995)	Adventure|Children's|Fantasy
2	3	Grumpier Old Men (1995)	Comedy|Romance
3	4	Waiting to Exhale (1995)	Comedy|Drama
4	5	Father of the Bride Part II (1995)	Comedy

Build model

[7]:
```
# Split once by train and test to demonstrate how different metrics work
split_dt = pd.Timestamp("2003-02-01")
df_train = ratings.loc[ratings["datetime"] < split_dt]
df_test = ratings.loc[ratings["datetime"] >= split_dt]
```

[8]:
```
%%time

# Prepare dataset, fit model and generate recommendations
dataset = Dataset.construct(df_train)
model = ImplicitItemKNNWrapperModel(TFIDFRecommender(K=10))
model.fit(dataset)
recos = model.recommend(
    users=ratings[Columns.User].unique(),
    dataset=dataset,
    k=10,
    filter_viewed=True,
)
```

```
CPU times: user 4.77 s, sys: 257 ms, total: 5.02 s
Wall time: 1.31 s
```

Calculate metrics

Metrics initialization

To calculate a metric it is necessary to create its object.

Most metrics have `k` parameter - the number of top recommendations that will be used for metric calculation.

Some metrics have additional parameters.

Simple metrics

[7]:
```
precision = Precision(k=10)
accuracy_1 = Accuracy(k=1)
accuracy_10 = Accuracy(k=10)
serendipity = Serendipity(k=10)
```

Metric with simple additional parameter

[8]:
```
ndcg = NDCG(k=10, log_base=3)
```

Metric with complex additional parameter

To calculate any diversity metric (e.g. `IntraListDivirsity`) you need to measure distance between items.

For example, you can use Hamming distance.

As features, let's use movie genres.

[9]:
```
movies["genre"] = movies["genres"].str.split("|")
genre_exploded = movies[["item_id", "genre"]].set_index("item_id").explode("genre")
genre_dummies = pd.get_dummies(genre_exploded, prefix="", prefix_sep="").groupby("item_id
↪").sum()
genre_dummies.head()
```

[9]:

item_id	Action	Adventure	Animation	Children's	Comedy	Crime	Documentary	Drama	Fantasy	Film-Noir	Horror	Musical	Mystery	Romance	Sci-Fi	Thriller	War	Western
1	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	1	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0
4	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0

[10]:
```
distance_calculator = PairwiseHammingDistanceCalculator(genre_dummies)
ild = IntraListDiversity(k=10, distance_calculator=distance_calculator)
```

Single metric calculation

The easiest way to calculate metric is to use `calc` method.

Every metric has it, but arguments are different.

If you need to get metric value for every user, use `calc_per_user` method.

[11]:
```
precision_value = precision.calc(reco=recos, interactions=df_test)
print(f"precision: {precision_value}")

precision_per_user = precision.calc_per_user(reco=recos, interactions=df_test)
print("\nprecision per user:")
display(precision_per_user.head())

print("Values are equal? ", precision_per_user.mean() == precision_value)
```

```
precision: 0.06464646464646465

precision per user:
```

```
user_id
195    0.3
229    0.0
343    0.0
349    0.0
398    0.5
dtype: float64
```

```
Values are equal?  True
```

[12]:
```
# Catalog is a set of items that we recommend.
# Sometimes not all items from train dataset appear in recommendations list.
catalog = df_train[Columns.Item].unique()
print("Accuracy@1: ", accuracy_1.calc(reco=recos, interactions=df_test, catalog=catalog))
print("Accuracy@10: ", accuracy_10.calc(reco=recos, interactions=df_test,␣
↪catalog=catalog))
```

```
Accuracy@1:  0.9956908534890186
Accuracy@10:  0.9935730756022174
```

[13]:
```
serendipity_value = serendipity.calc(
    reco=recos,
    interactions=df_test,
    prev_interactions=df_train,
    catalog=catalog
)
print("Serendipity: ", serendipity_value)
```

```
Serendipity:  2.3436131849908687e-05
```

[14]:
```
print("NDCG: ", ndcg.calc(reco=recos, interactions=df_test))
```

```
NDCG:  0.06808226116073855
```

[15]:
```
%%time
print("ILD: ", ild.calc(reco=recos))
```

```
ILD:  3.1908278145695363
CPU times: user 2.1 s, sys: 556 ms, total: 2.66 s
Wall time: 2.64 s
```

Multiple metrics calculation

It is possible to calculate a bunch of metrics using only one function - `calc_metrics`.

It contains same optimisations in performance: if several metrics do the same calculations, they will be performed only once.

[16]:
```
metrics = {
    "precision": precision,
    "accuracy@1": accuracy_1,
    "accuracy@10": accuracy_10,
    "ndcg": ndcg,
    "serendipity": serendipity,
    "diversity": ild,
}

# Some arguments can be omitted if they are not needed for metrics calculation
calc_metrics(
    metrics,
    reco=recos,
    interactions=df_test,
    prev_interactions=df_train,
    catalog=catalog
)
```

[16]:
```
{'precision': 0.06464646464646465,
 'accuracy@10': 0.9935730756022174,
 'accuracy@1': 0.9956908534890186,
 'ndcg': 0.06808226116073855,
 'diversity': 3.1908278145695363,
 'serendipity': 2.3436131849908687e-05}
```

[ ]:
```

```

3.9.4 Examples of constructing datasets with features in RecTools

Some models allow using explicit user (sex, age, etc.) and item (genre, year, ...) features. Let's see how we can process them to RecTools dataset.

After creating the dataset, training models with features is as simple as `model.fit(dataset_with_features)`

[2]:
```
import os
import threadpoolctl

import numpy as np
import pandas as pd
from implicit.als import AlternatingLeastSquares
```

(continues on next page)

(continued from previous page)

```
from rectools import Columns
from rectools.dataset import Dataset
from rectools.models import ImplicitALSWrapperModel

# For implicit ALS
os.environ["OPENBLAS_NUM_THREADS"] = "1"
threadpoolctl.threadpool_limits(1, "blas")
```

Load data: Movielens 1m

[27]:
```
%%time
!wget -q https://files.grouplens.org/datasets/movielens/ml-1m.zip -O ml-1m.zip
!unzip -o ml-1m.zip
!rm ml-1m.zip
```

```
Archive:  ml-1m.zip
  inflating: ml-1m/movies.dat
  inflating: ml-1m/ratings.dat
  inflating: ml-1m/README
  inflating: ml-1m/users.dat
CPU times: user 43.2 ms, sys: 62.3 ms, total: 106 ms
Wall time: 3.11 s
```

[28]:
```
%%time
ratings = pd.read_csv(
    "ml-1m/ratings.dat",
    sep="::",
    engine="python",  # Because of 2-chars separators
    header=None,
    names=[Columns.User, Columns.Item, Columns.Weight, Columns.Datetime],
)
print(ratings.shape)
ratings.head()
```

```
(1000209, 4)
CPU times: user 3.84 s, sys: 357 ms, total: 4.2 s
Wall time: 4.17 s
```

[28]:

	user_id	item_id	weight	datetime
0	1	1193	5	978300760
1	1	661	3	978302109
2	1	914	3	978301968
3	1	3408	4	978300275
4	1	2355	5	978824291

[29]:
```
%%time
users = pd.read_csv(
    "ml-1m/users.dat",
    sep="::",
    engine="python",  # Because of 2-chars separators
    header=None,
```

(continues on next page)

(continued from previous page)

```
    names=[Columns.User, "sex", "age", "occupation", "zip_code"],
)
print(users.shape)
users.head()
```

```
(6040, 5)
CPU times: user 17.2 ms, sys: 2.38 ms, total: 19.6 ms
Wall time: 18.8 ms
```

[29]:

	user_id	sex	age	occupation	zip_code
0	1	F	1	10	48067
1	2	M	56	16	70072
2	3	M	25	15	55117
3	4	M	45	7	02460
4	5	M	25	20	55455

[30]:
```
# Select only users that present in 'ratings' table
users = users.loc[users["user_id"].isin(ratings["user_id"])].copy()
```

Data types: categorical and numerical

Generally there are 2 kind of features in data: categorical and numerical. For classic recommender algorithms categorical features are usually one-hot-encoded and stored in sparse format. Numerical features can be used in the original form (e.g. processed by MinMaxScaler), but they can also be binarized, transformed to categorical and then one-hot encoded.

Depending on your data you can select to store features in `sparse` or `dense` format within RecTools dataset. `dense` format requires all features to be numerical. `sparse` format doesn't have any constraints and can include numerical features as well.

During training RecTools models will transform features to the format that is apllicable. iALS with features will transform feature to `dense` format. LightFM and DSSM will transform to `sparse`. All of these transformations happen under the hood and no values are actually affected.

Now let's see processing routines.

Features storage: Sparse example

For `sparse` format we need to create a dataframe in flatten format with columns `id`, `feature`, `value`. This way we can have any number of entries for each feature for any user (ot item). This is often the case for movie genres for example (one movie has 5 genres).

[31]:
```
# Let's prepare a flatten dataframe with 3 user features
user_features_frames = []
for feature in ["sex", "age", "occupation"]:
    feature_frame = users.reindex(columns=["user_id", feature])
    feature_frame.columns = ["id", "value"]
    feature_frame["feature"] = feature
    user_features_frames.append(feature_frame)
user_features = pd.concat(user_features_frames)
```

[32]:
```
# Let's see how this looks for users `1` and `2`
user_features.query("id in [1, 2]").sort_values("id")
```

[32]:
```
   id value     feature
0   1     F         sex
0   1     1         age
0   1    10  occupation
1   2     M         sex
1   2    56         age
1   2    16  occupation
```

[33]:
```
# Now we construct the dataset
sparse_features_dataset = Dataset.construct(
    ratings,
    user_features_df=user_features,  # our flatten dataframe
    cat_user_features=["sex", "age"], # these will be one-hot-encoded. All other␣
↪features must be numerical already
    make_dense_user_features=False  # for `sparse` format
)
```

In this dataset user features are now stored in `sparse` format.

`cat_user_features` have all their possible values retrieved, one-hot-encoded and stored in sparse matrix.

All other features (`direct`) have their values stored in the same sparse matrix (one columns for one direct feature). Here we make "occupation" a direct feature just for a quick example on data storage. It actually has categorical nature.

Rows of the sparse matrix correspond to internal user ids in dataset. Which are identical to row numbers in ui_csr matrix which is used for model training in most of the recommender models.

Let's look inside the dataset to check how the data is stored

[34]:
```
# storing format for features
sparse_features_dataset.user_features.values
```

[34]:
```
<6040x10 sparse matrix of type '<class 'numpy.float32'>'
        with 18120 stored elements in Compressed Sparse Row format>
```

[35]:
```
# feature names and values (sparse matrix columns)
sparse_features_dataset.user_features.names
```

[35]:
```
(('occupation', '__is_direct_feature'),
 ('sex', 'F'),
 ('sex', 'M'),
 ('age', 1),
 ('age', 56),
 ('age', 25),
 ('age', 45),
 ('age', 50),
 ('age', 35),
 ('age', 18))
```

[36]:
```
# example of stored features for 5 users
sparse_features_dataset.user_features.values[:5].toarray()
```

[36]:
```
array([[10.,  1.,  0.,  1.,  0.,  0.,  0.,  0.,  0.,  0.],
       [16.,  0.,  1.,  0.,  1.,  0.,  0.,  0.,  0.,  0.],
       [15.,  0.,  1.,  0.,  0.,  1.,  0.,  0.,  0.,  0.],
```

(continues on next page)

(continued from previous page)

```
       [ 7.,  0.,  1.,  0.,  0.,  0.,  1.,  0.,  0.,  0.],
       [20.,  0.,  1.,  0.,  0.,  1.,  0.,  0.,  0.,  0.]], dtype=float32)
```

Features storage: Dense example

Now let's create a dataset with `dense` features.

We need a classic dataframe with one column for each feature and one row for each subject (user or item).

Important: All feature values must be numeric

Important: You must set features for all objects (users or items). If you do not have some feature for some user (item) then use any method (zero, mean value, etc.) to fill it.

[37]:
```
user_numeric_features = users[[Columns.User, "age", "occupation"]]
user_numeric_features.head()
```

[37]:
```
   user_id  age  occupation
0        1    1          10
1        2   56          16
2        3   25          15
3        4   45           7
4        5   25          20
```

[38]:
```
dense_features_dataset = Dataset.construct(
    ratings,
    user_features_df=user_numeric_features,
    make_dense_user_features=True  # for `dense` format
)
```

Let's look how the data is stored now. This is a 2-d numpy array. Row numbers correspond to internal user ids in dataset.

[39]:
```
# feature names (array columns)
dense_features_dataset.user_features.names
```

[39]:
```
('age', 'occupation')
```

[40]:
```
# example of stored features for 5 users
dense_features_dataset.user_features.values[:5]
```

[40]:
```
array([[ 1., 10.],
       [56., 16.],
       [25., 15.],
       [45.,  7.],
       [25., 20.]], dtype=float32)
```

Feeding features to models

Now we can just fit model using prepared dataset. For this we choose models that have support for using features in training (e.g. iALS, LightFM, DSSM, PopularInCategory).

[41]:
```
model = ImplicitALSWrapperModel(AlternatingLeastSquares(10, num_threads=32))
model.fit(dense_features_dataset)
```

```
100%|| 1/1 [00:00<00:00, 17.08it/s]
```

[41]:
```
<rectools.models.implicit_als.ImplicitALSWrapperModel at 0x7f6fe355bf10>
```

[42]:
```
model = ImplicitALSWrapperModel(AlternatingLeastSquares(10, num_threads=32))
model.fit(sparse_features_dataset)
```

```
/data/home/dmtikhono1/git_project/RecTools/rectools/dataset/features.py:399: UserWarning:
↪ Converting sparse features to dense array may cause MemoryError
  warnings.warn("Converting sparse features to dense array may cause MemoryError")
100%|| 1/1 [00:00<00:00, 12.94it/s]
```

[42]:
```
<rectools.models.implicit_als.ImplicitALSWrapperModel at 0x7f6fe355be10>
```

Final notes

- If model requires features in a specific format, it will convert them under the hood. This is why we can get a warning, fitting iALS with sparse features. Model fits anyway, just remember about possible memory problems
- LightFM and DSSM prefer one-hot-encoded features. So it is a good idea to binarize all direct features and make them categorical. But you can also try to apply MinMaxScaler to direct values.
- iALS works good with both direct and categorical features. Direct features can be MinMaxScaled
- PopularInCategory requires `sparse` features and a selected category because of its nature

3.9.5 Benchmark quality: RecTools wrapper for iALS with features

- Load and preprocess Kion dataset
- Grid Search for best pure iALS model params (classic algorithm with no features)
- Train 3 best params models with different feature selections (all features / user features / item features / biases)
- Visualize ans sumarize results
- Technical details on RecTools implementation of iALS with features

Important! Grid search takes considerable time. Please keep in mind if you want to run this notebook

[ ]:
```
import os
import threadpoolctl
import warnings
from pathlib import Path

import pandas as pd
import numpy as np
import seaborn as sns
from matplotlib import pyplot as plt
```

(continues on next page)

(continued from previous page)

```
from rectools.metrics import MAP, calc_metrics, MeanInvUserFreq, Serendipity
from rectools.models import ImplicitALSWrapperModel
from rectools import Columns
from rectools.dataset import Dataset
from implicit.als import AlternatingLeastSquares

warnings.filterwarnings('ignore')
sns.set_theme(style="whitegrid")

# For implicit ALS
os.environ["OPENBLAS_NUM_THREADS"] = "1"
threadpoolctl.threadpool_limits(1, "blas")
```

Load and preprocess data: Kion

[69]:
```
%%time
!wget -q https://github.com/irsafilo/KION_DATASET/raw/
↪f69775be31fa5779907cf0a92ddedb70037fb5ae/data_original.zip -O data_original.zip
!unzip -o data_original.zip
!rm data_original.zip
```

```
Archive:  data_original.zip
   creating: data_original/
  inflating: data_original/interactions.csv
  inflating: __MACOSX/data_original/._interactions.csv
  inflating: data_original/users.csv
  inflating: __MACOSX/data_original/._users.csv
  inflating: data_original/items.csv
  inflating: __MACOSX/data_original/._items.csv
CPU times: user 0 ns, sys: 473 ms, total: 473 ms
Wall time: 8.05 s
```

[79]:
```
DATA_PATH = Path("data_original")

users = pd.read_csv(DATA_PATH / 'users.csv')
items = pd.read_csv(DATA_PATH / 'items.csv')
interactions = (
    pd.read_csv(DATA_PATH / 'interactions.csv', parse_dates=["last_watch_dt"])
    .rename(columns={"last_watch_dt": Columns.Datetime})
)
```

[4]:
```
# Process interactions
interactions[Columns.Weight] = np.where(interactions['watched_pct'] > 10, 3, 1)

# Split to train / test
max_date = interactions[Columns.Datetime].max()
train = interactions[interactions[Columns.Datetime] < max_date - pd.Timedelta(days=7)].
↪copy()
test = interactions[interactions[Columns.Datetime] >= max_date - pd.Timedelta(days=7)].
```

(continues on next page)

(continued from previous page)

```
↪copy()
train.drop(train.query("total_dur < 300").index, inplace=True)
cold_users = set(test[Columns.User]) - set(train[Columns.User])
test.drop(test[test[Columns.User].isin(cold_users)].index, inplace=True)
test_users = test[Columns.User].unique()
catalog=train[Columns.Item].unique()

# Process user features to the form of a flatten dataframe
users.fillna('Unknown', inplace=True)
users = users.loc[users[Columns.User].isin(train[Columns.User])].copy()
user_features_frames = []
for feature in ["sex", "age", "income"]:
    feature_frame = users.reindex(columns=[Columns.User, feature])
    feature_frame.columns = ["id", "value"]
    feature_frame["feature"] = feature
    user_features_frames.append(feature_frame)
user_features = pd.concat(user_features_frames)

# Process item features to the form of a flatten dataframe
items = items.loc[items[Columns.Item].isin(train[Columns.Item])].copy()
items["genre"] = items["genres"].str.lower().str.replace(", ", ",", regex=False).str.
↪split(",")
genre_feature = items[["item_id", "genre"]].explode("genre")
genre_feature.columns = ["id", "value"]
genre_feature["feature"] = "genre"
content_feature = items.reindex(columns=[Columns.Item, "content_type"])
content_feature.columns = ["id", "value"]
content_feature["feature"] = "content_type"
item_features = pd.concat((genre_feature, content_feature))
```

[247]:
```
user_features.head()
```

[247]:

	id	value	feature
0	973171		sex
1	962099		sex
3	721985		sex
4	704055		sex
5	1037719		sex

[248]:
```
item_features.head()
```

[248]:

	id	value	feature
0	10711		genre
0	10711	genre	
0	10711	genre	
0	10711	genre	
1	2508	genre	

Datasets

Prepare datasets with different feature selection (no_features / full_features / users only / items only)

[147]:
```
dataset_no_features = Dataset.construct(
    interactions_df=train,
)

dataset_full_features = Dataset.construct(
    interactions_df=train,
    user_features_df=user_features,
    cat_user_features=["sex", "age", "income"],
    item_features_df=item_features,
    cat_item_features=["genre", "content_type"],
)

dataset_item_features = Dataset.construct(
    interactions_df=train,
    item_features_df=item_features,
    cat_item_features=["genre", "content_type"],
)

dataset_user_features = Dataset.construct(
    interactions_df=train,
    user_features_df=user_features,
    cat_user_features=["sex", "age", "income"],
)

feature_datasets = {
    "full_features": dataset_full_features,
    "item_features": dataset_item_features,
    "user_features": dataset_user_features
}
```

Add dataset which allows iALS to learn user and item biases. For this we create for all users and items constant feature with value 1

[148]:
```
# Prepare dataset with biases as features

item_biases = pd.DataFrame({
    "id": catalog,
    "bias": 1
})
user_biases = pd.DataFrame({
    "id": train[Columns.User].unique(),
    "bias": 1
})

dataset_with_biases = Dataset.construct(
    interactions_df=train,
    user_features_df=user_biases,
    make_dense_user_features=True,
    item_features_df=item_biases,
    make_dense_item_features=True
```

(continues on next page)

(continued from previous page)

```
)
feature_datasets["biases"] = dataset_with_biases
```

Metrics

[6]:
```
metrics_name = {
    'MAP': MAP,
    'MIUF': MeanInvUserFreq,
    'Serendipity': Serendipity


}
metrics = {}
for metric_name, metric in metrics_name.items():
    for k in (1, 5, 10):
        metrics[f'{metric_name}@{k}'] = metric(k=k)
```

Model initialization

[44]:
```
K_RECOS = 10
NUM_THREADS = 32
RANDOM_STATE = 32
ITERATIONS = 10
```

[45]:
```
def make_base_model(factors: int, regularization: float, alpha: float, fit_features_
↪together: bool=False):
    return ImplicitALSWrapperModel(
        AlternatingLeastSquares(
            factors=factors,
            regularization=regularization,
            alpha=alpha,
            random_state=RANDOM_STATE,
            use_gpu=False,
            num_threads = NUM_THREADS,
            iterations=ITERATIONS),
        fit_features_together = fit_features_together,
        )
```

Grid Search for best pure iALS model params

[46]:
```
alphas = [1, 10, 100]
regularizations = [0.01, 0.1, 0.5]
factors = [32, 64, 128]
```

[ ]:
```
results = []
dataset = dataset_no_features
```

(continues on next page)

(continued from previous page)

```
for alpha in alphas:
    for regularization in regularizations:
        for n_factors in factors:
            model_name = f"no_features_factors_{n_factors}_alpha_{alpha}_reg_
↪{regularization}"
            model = make_base_model(factors=n_factors, regularization=regularization,␣
↪alpha=alpha)
            model.fit(dataset)
            recos = model.recommend(
                users=test_users,
                dataset=dataset,
                k=K_RECOS,
                filter_viewed=True,
            )
            metric_values = calc_metrics(metrics, recos, test, train, catalog)
            metric_values["model"] = model_name
            results.append(metric_values)
```

[145]:
```
pure_df = pd.DataFrame(results).set_index("model").sort_values(by=["MAP@10",
↪"Serendipity@10"], ascending=False)
pure_df.head(5)
```

[145]:

model	MAP@1	MAP@5	MAP@10	\
no_features_factors_32_alpha_10_reg_0.5	0.023853	0.044661	0.050860	
no_features_factors_32_alpha_10_reg_0.01	0.023117	0.042483	0.049064	
no_features_factors_32_alpha_10_reg_0.1	0.022661	0.041494	0.048325	
no_features_factors_64_alpha_10_reg_0.01	0.020317	0.036539	0.041775	
no_features_factors_64_alpha_10_reg_0.5	0.020408	0.036451	0.041736	

model	MIUF@1	MIUF@5	MIUF@10	\
no_features_factors_32_alpha_10_reg_0.5	5.122939	5.864645	6.319260	
no_features_factors_32_alpha_10_reg_0.01	5.140355	5.880019	6.311438	
no_features_factors_32_alpha_10_reg_0.1	5.220894	5.960500	6.353801	
no_features_factors_64_alpha_10_reg_0.01	6.010125	6.590613	6.917891	
no_features_factors_64_alpha_10_reg_0.5	6.095905	6.655192	6.991212	

model	Serendipity@1	Serendipity@5	\
no_features_factors_32_alpha_10_reg_0.5	0.000107	0.000126	
no_features_factors_32_alpha_10_reg_0.01	0.000106	0.000125	
no_features_factors_32_alpha_10_reg_0.1	0.000109	0.000128	
no_features_factors_64_alpha_10_reg_0.01	0.000219	0.000206	
no_features_factors_64_alpha_10_reg_0.5	0.000232	0.000212	

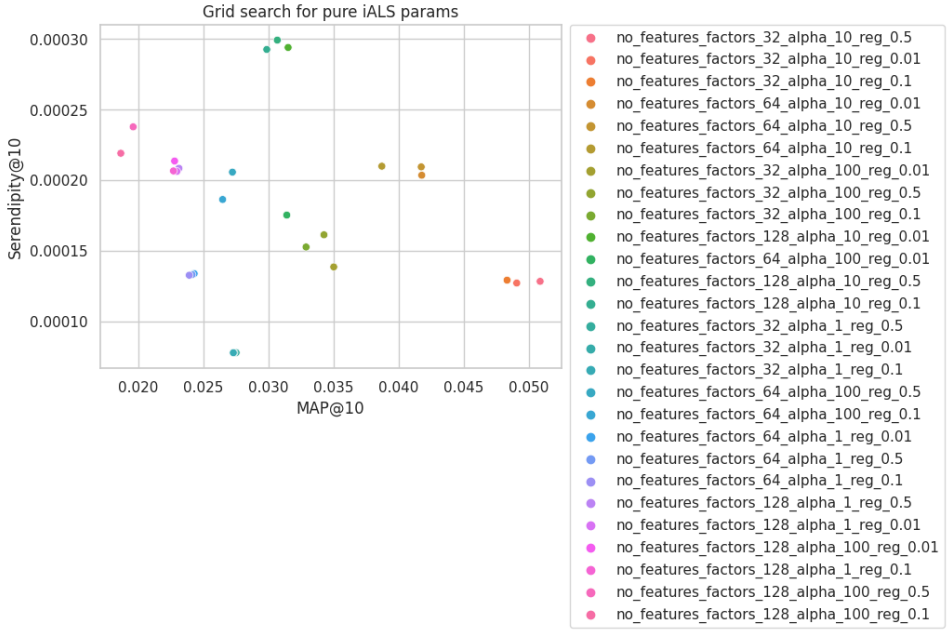
model	Serendipity@10	with_features
no_features_factors_32_alpha_10_reg_0.5	0.000128	False
no_features_factors_32_alpha_10_reg_0.01	0.000127	False
no_features_factors_32_alpha_10_reg_0.1	0.000129	False

(continues on next page)

(continued from previous page)

```
no_features_factors_64_alpha_10_reg_0.01        0.000203          False
no_features_factors_64_alpha_10_reg_0.5         0.000209          False
```

[250]:
```
sns.scatterplot(data = pure_df, x="MAP@10", y="Serendipity@10", hue="model", legend=True)
plt.legend(bbox_to_anchor=(1.02, 1), loc='upper left', borderaxespad=0)
plt.title("Grid search for pure iALS params")
plt.show()
```



nbsphinx-code-borderwhite

We achieved maximum MAP@10: **0.05**.

From the plot we can clearly see that trying to achieve higher Serendipity metric leads to decrease in MAP. This is a classical trade-off. Lets' see if we can increase both metrics simultaneously, introducing features to the model

Lets' see best models params:

[143]:
```
# Best params for MAP@10
pure_df.set_index("model").sort_values("MAP@10").tail(1)
```

[143]:
```
                                          MAP@1     MAP@5   MAP@10  \
model
no_features_factors_32_alpha_10_reg_0.5  0.023853  0.044661  0.05086

                                           MIUF@1    MIUF@5  MIUF@10  \
model
no_features_factors_32_alpha_10_reg_0.5  5.122939  5.864645  6.31926

                                         Serendipity@1  Serendipity@5  \
model
no_features_factors_32_alpha_10_reg_0.5       0.000107       0.000126

                                         Serendipity@10
model
no_features_factors_32_alpha_10_reg_0.5        0.000128
```

[144]:
```
# Best params for Serendipity@10
pure_df.set_index("model").sort_values("Serendipity@10").tail(1)
```

[144]:
```
                                                MAP@1     MAP@5    MAP@10  \
model
no_features_factors_128_alpha_10_reg_0.5  0.015226  0.026761  0.030691

                                                MIUF@1    MIUF@5   MIUF@10  \
model
no_features_factors_128_alpha_10_reg_0.5  7.189521  7.565286  7.750308

                                          Serendipity@1  Serendipity@5  \
model
no_features_factors_128_alpha_10_reg_0.5       0.000412       0.000338

                                          Serendipity@10
model
no_features_factors_128_alpha_10_reg_0.5        0.000299
```

Here we can see that both max MAP and max Serendipity models have the same params for alpha and regularization. Let's fix them but keep all dimension size options for further research. This way we have 3 best models from pure iALS and try to add features to them.

Add different feature selections to 3 best iALS models

[171]:
```
# Best grid search params for pure iALS models
factors_options = (32, 64, 128)
ALPHA = 10
REG = 0.5

# We have two options for training iALS with features in RecTools
fit_features_together = (True, False)

# We have datasets with different feature selections
feature_datasets.keys()
```

[171]:
```
dict_keys(['full_features', 'item_features', 'user_features', 'biases'])
```

[ ]:
```
features_results = []
for dataset_name, dataset in feature_datasets.items():
    for n_factors in factors_options:
        for features_option in fit_features_together:
            model_name = f"{dataset_name}_factors_{n_factors}_fit_together_{features_
↪option}"
            model = make_base_model(factors = n_factors, regularization=REG, alpha=ALPHA,
↪ fit_features_together=features_option)
            model.fit(dataset)
            recos = model.recommend(
                users=test_users,
                dataset=dataset,
                k=K_RECOS,
                filter_viewed=True,
```

(continues on next page)

(continued from previous page)

```
            )
            metric_values = calc_metrics(metrics, recos, test, train, catalog)
            metric_values["model"] = model_name
            features_results.append(metric_values)
```

[244]:
```
features_df = (
    pd.DataFrame(features_results)
    .set_index("model")
    .sort_values(by=["MAP@10", "Serendipity@10"], ascending=False)
)
features_df.head(5)
```

[244]:

model	MAP@1	MAP@5	MAP@10	\
full_features_factors_128_fit_together_True	0.040123	0.069483	0.076887	
full_features_factors_64_fit_together_True	0.040797	0.069343	0.076602	
full_features_factors_32_fit_together_True	0.040904	0.069000	0.076101	
biases_factors_128_fit_together_False	0.035941	0.067140	0.074253	
biases_factors_64_fit_together_False	0.035555	0.064373	0.072245	

model	MIUF@1	MIUF@5	MIUF@10	\
full_features_factors_128_fit_together_True	3.658149	4.369860	5.116690	
full_features_factors_64_fit_together_True	3.599685	4.269495	4.986737	
full_features_factors_32_fit_together_True	3.492285	4.162145	4.882946	
biases_factors_128_fit_together_False	3.600921	3.945118	4.577787	
biases_factors_64_fit_together_False	3.479163	4.028714	4.648506	

model	Serendipity@1	Serendipity@5	\
full_features_factors_128_fit_together_True	0.000042	0.000049	
full_features_factors_64_fit_together_True	0.000035	0.000040	
full_features_factors_32_fit_together_True	0.000029	0.000036	
biases_factors_128_fit_together_False	0.000067	0.000076	
biases_factors_64_fit_together_False	0.000029	0.000045	

model	Serendipity@10
full_features_factors_128_fit_together_True	0.000055
full_features_factors_64_fit_together_True	0.000046
full_features_factors_32_fit_together_True	0.000039
biases_factors_128_fit_together_False	0.000088
biases_factors_64_fit_together_False	0.000061

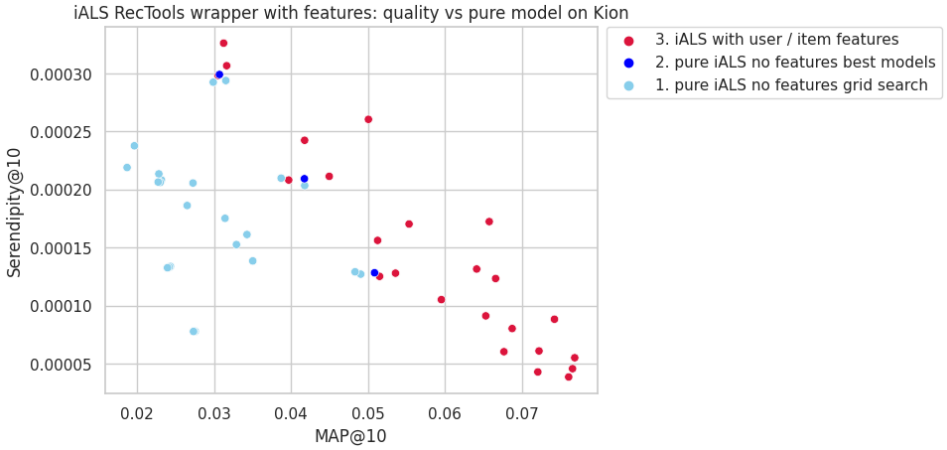
Let's visualise all of our research on the same plot to summarize results

Visualise all results

[237]:
```
# prepare info for summary plot
no_features_best_models = [
    "no_features_factors_128_alpha_10_reg_0.5",
    "no_features_factors_64_alpha_10_reg_0.5",
    "no_features_factors_32_alpha_10_reg_0.5"
]

pure_df["step"] = "1. pure iALS no features grid search"
pure_df.loc[pure_df.index.isin(no_features_best_models), "step"] = "2. pure iALS no␣
↪features best models"
features_df["step"] = "3. iALS with user / item features"
all_df = pd.concat([features_df, pure_df])
```

[246]:
```
sns.scatterplot(data = all_df, x="MAP@10", y="Serendipity@10", hue="step",
                palette=["crimson", "blue", "skyblue"])
plt.legend(bbox_to_anchor=(1.02, 1), loc='upper left', borderaxespad=0)
plt.title("iALS RecTools wrapper with features: quality vs pure model on Kion")
plt.show()
```



nbsphinx-code-borderwhite

Summary on our benchmark

- Almost all of our feature selection options increased quality of the pure iALS best models.
- With features we achieved maximum for MAP@10 `0.077`! **This is a 50% increase from pure model best result**.
- We didn't simply increase MAP trading off Serendipity. Instead many of our feature options raised both metrics simultanuosly, which is a strong evidence of better performing algorithms.

Important notes: - We skipped tuning optimal weights of interactions for iALS training - We skipped tuning number of iterations for iALS training - We didn't preprocess features - We didn't do actual feature selection and tested only quick subsets of features - We didn't tune params again after adding features to training

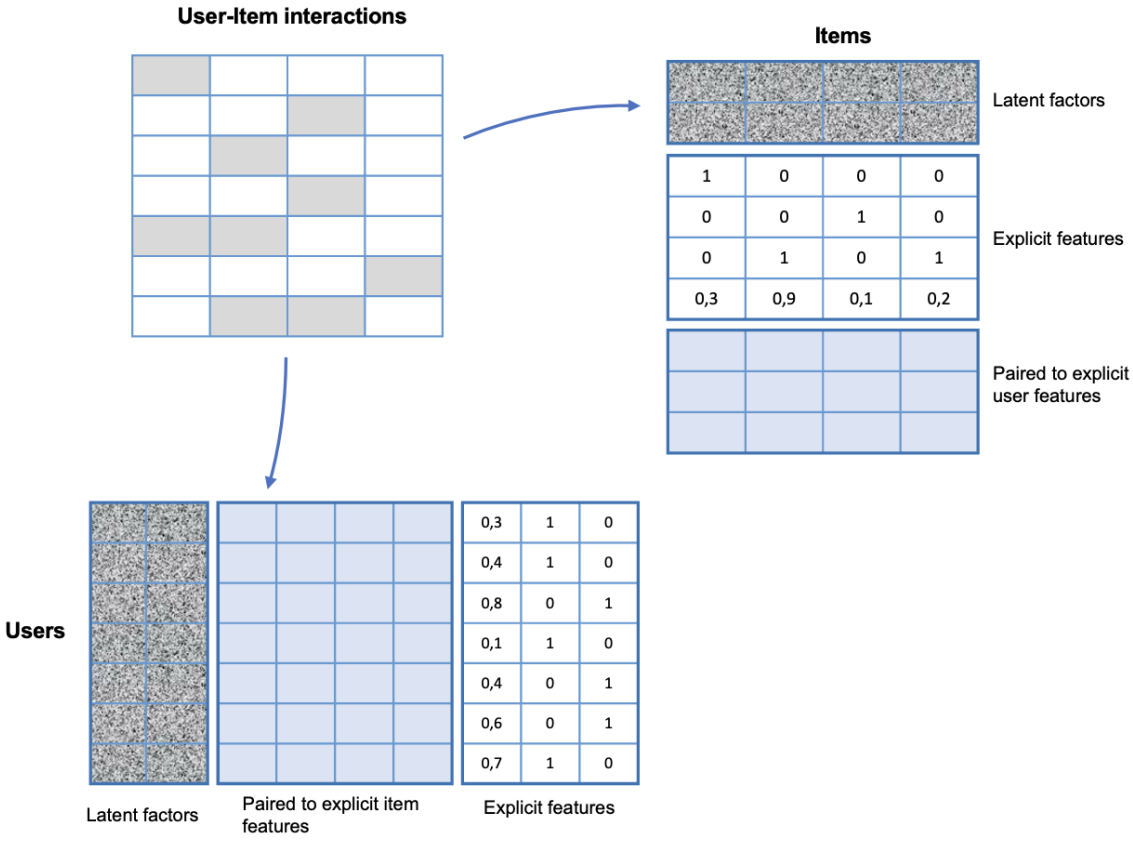
Technical details on RecTools implementation of iALS with features

In pure iALS model latent factors are initialized with random values.

When we train RecTools iALS wrapper on dataset with features, we add explicit user and/or item features to embeddings. After adding some number of item explicit features to the matrix of item embeddings, we also add the same number of columns to user embeddings matrix. These added columns are user factors paired to explicit item factors. We do the same for explicit user factors and item factors paired to them.

Technical moments: - Paired factors are trained. Explicit features remain untouched - We can train paried factors together with latent factors (`fit_features_together=True`) or after training latent factors (`fit_features_together=False`) - Explicit features are converted to `dense` format before training. Remember to keep reasonable amount of data to avoid memory problems - Training and inference is proceeded with `implicit` library methods

Full scheme:



3.9.6 Benchmark inference speed: RecTools wrapper for LightFM

Comparing speed of inference: RecTools LightFM wrapper vs original LightFM framework. Inference on Movielens 20-m dataset

Real speed may vary depending on actual hardware and library versions

LightFM library is required to run this notebook. You can install it with `pip install rectools[lightfm]`

[2]:
```
import rectools
import lightfm
print(rectools.__version__)
print(lightfm.__version__)
```

```
0.4.0
1.17
```

[ ]:
```
import os
import warnings
import time

from pathlib import Path
from pprint import pprint

import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

from rectools import Columns
from rectools.dataset import Dataset
from rectools.models import LightFMWrapperModel

from lightfm import LightFM

os.environ["OPENBLAS_NUM_THREADS"] = "1"
sns.set_theme(style="whitegrid")
```

Load data: Movielens 20m

[3]:
```
%%time
!wget -q https://files.grouplens.org/datasets/movielens/ml-20m.zip -O ml-20m.zip
!unzip -o ml-20m.zip
!rm ml-20m.zip
```

```
Archive:  ml-20m.zip
  inflating: ml-20m/genome-scores.csv
  inflating: ml-20m/genome-tags.csv
  inflating: ml-20m/links.csv
  inflating: ml-20m/movies.csv
  inflating: ml-20m/ratings.csv
  inflating: ml-20m/README.txt
  inflating: ml-20m/tags.csv
```

(continues on next page)

(continued from previous page)

```
CPU times: user 266 ms, sys: 105 ms, total: 371 ms
Wall time: 32.5 s
```

[4]:
```
ratings = pd.read_csv(
    "ml-20m/ratings.csv",
    header=0,
    names=[Columns.User, Columns.Item, Columns.Weight, Columns.Datetime],
)
print(ratings.shape)
ratings.head()
```

```
(20000263, 4)
```

[4]:

	user_id	item_id	weight	datetime
0	1	2	3.5	1112486027
1	1	29	3.5	1112484676
2	1	32	3.5	1112484819
3	1	47	3.5	1112484727
4	1	50	3.5	1112484580

Train LightFM model with different latent factors dimension size

[5]:
```
def make_base_model(factors: int):
    return LightFMWrapperModel(LightFM(no_components=factors, loss="bpr"))

dataset = Dataset.construct(ratings)

fitted_models = {}

factors = list(range(64, 257, 64))
for n_factors in factors:
    model = make_base_model(n_factors)
    model.fit(dataset)
    fitted_models[n_factors] = model
```

Compute inference speed for both frameworks on all models

[6]:
```
# we will use 8 threads for both RecTools and LightFM inference
# we will run each experiment 10 times to get mean values
NUM_THREADS = 8
K_RECOS = 10
NUM_REPEATS = 10
```

[7]:
```
# prepare external (real) movielens user ids for RecTools inference
train_users = dataset.user_id_map.external_ids

# prepare internal sparse matrix user and items ids for LightFM inference
user_ids = dataset.user_id_map.internal_ids
item_ids = dataset.item_id_map.internal_ids
```

(continues on next page)

(continued from previous page)

```
n_users = len(user_ids)
n_items = len(item_ids)

assert len(train_users) == len(user_ids)
print(len(train_users))
```

```
138493
```

[8]:
```
def benchmark_rectools(models, n_users):
    results = []
    users = train_users[:n_users]
    for run in range(NUM_REPEATS):
        for n_factors, model in models.items():
            start = time.time()

            # RecTools inference. It is processed by batches of users internally
            model.n_threads = NUM_THREADS
            recos = model.recommend(
                users=users,
                dataset=dataset,
                k=K_RECOS,
                filter_viewed=False,
                add_rank_col=False
            )

            elapsed = time.time() - start
            results.append({"factors": n_factors, "framework": "rectools", "inference_
↪speed": elapsed, "run": run})
    return results
```

[9]:
```
def benchmark_lightfm_batch(models, n_users, batch_size = 1000):
    results = []
    users = user_ids[:n_users]
    for run in range(NUM_REPEATS):
        for n_factors, model in models.items():
            # get LightFM framework model from RecTools wrapper
            lightfm_model = model.model
            start = time.time()

            # LightFM inference. We proceed it by batches to avoid memory problems
            reco_by_batch = []
            n_batches = n_users // batch_size
            if n_users % batch_size > 0:
                n_batches += 1
            for i_batch in range(n_batches):
                batch_users_ids = users[i_batch * batch_size : (i_batch + 1) * batch_
↪size]
                batch_scores = lightfm_model.predict(
                    user_ids = np.repeat(batch_users_ids, n_items),
                    item_ids = np.tile(item_ids, len(batch_users_ids)),
                    num_threads = NUM_THREADS
                ).reshape(len(batch_users_ids), n_items)
```

(continues on next page)

(continued from previous page)

```
                # scores are not sorted so we need to sort them and get top K_RECOS␣
↪items for each user
                batch_unsorted_reco_positions = batch_scores.argpartition(-K_RECOS,␣
↪axis=1)[:, -K_RECOS:]
                batch_unsorted_reco_scores = np.take_along_axis(batch_scores, batch_
↪unsorted_reco_positions, axis=1)
                batch_recs = np.take_along_axis(
                        batch_unsorted_reco_positions, batch_unsorted_reco_scores.
↪argsort()[:, ::-1], axis=1
                    )
                reco_by_batch.append(batch_recs)
            reco_by_batch = np.vstack(reco_by_batch)

            elapsed = time.time() - start
            results.append({"factors": n_factors, "framework": "lightfm", "inference_
↪speed": elapsed, "run": run})
    return results
```

Benchmark frameworks

Be careful when running this code. Inference with all the repeats and settings for all 140k train users takes a lot of time.

We advice to select about 10k users for comparison on regular hardware and skip full comparison

[10]:
```
# inference for 10k users on Movielens-20m

N_USERS = 10000
print(f"Num users for inference: {N_USERS}")
rectools_results_ten_k = benchmark_rectools(fitted_models, n_users=N_USERS)
lightfm_results_batch_ten_k = benchmark_lightfm_batch(fitted_models, n_users=N_USERS)
results_ten_k = pd.DataFrame(lightfm_results_batch_ten_k + rectools_results_ten_k)
```

```
Num users for inference: 10000
```

[11]:
```
# inference for all train users on Movielens-20m

print(f"Num users for inference: {n_users}")
rectools_results = benchmark_rectools(fitted_models, n_users=n_users)
lightfm_results_batch = benchmark_lightfm_batch(fitted_models, n_users=n_users)
results_all = pd.DataFrame(lightfm_results_batch + rectools_results)
```

```
Num users for inference: 138493
```

[12]:
```
results_all.head()
```

[12]:

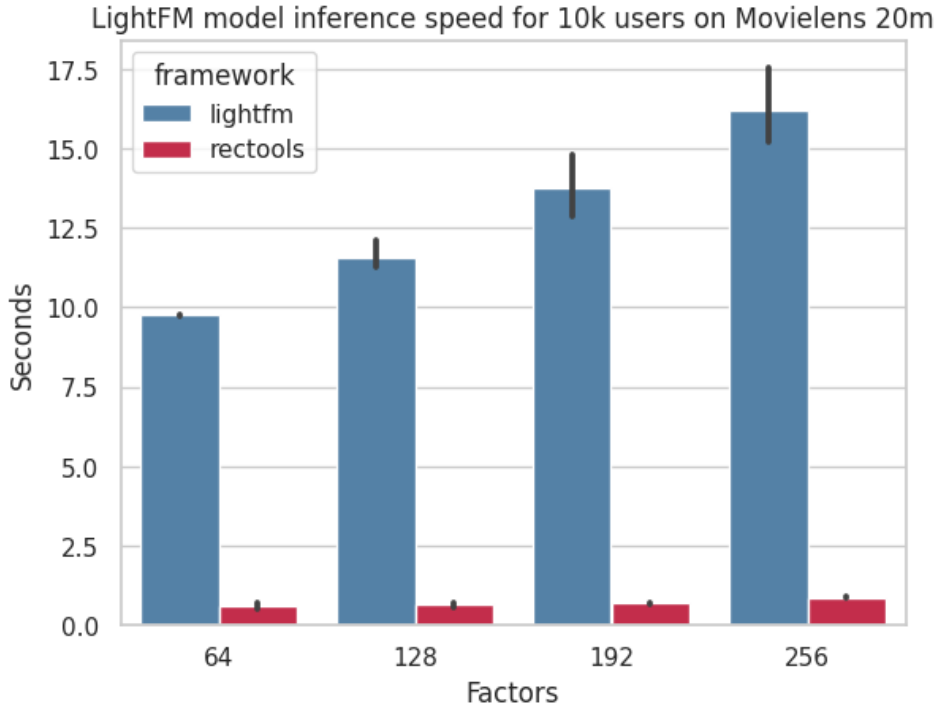
	factors	framework	inference_speed	run
0	64	lightfm	135.769374	0
1	128	lightfm	160.676028	0
2	192	lightfm	190.658902	0
3	256	lightfm	204.438340	0
4	64	lightfm	134.842623	1

Compare the difference

Inference speed

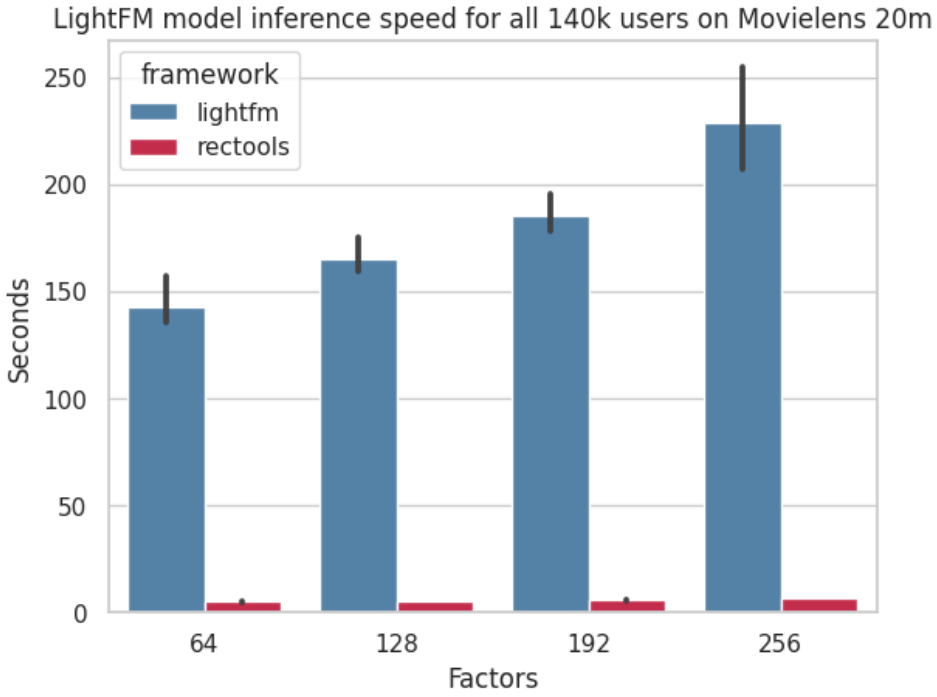
- RecTools wrapper has 5 to 25 times faster inference then original LightFM framework on the same number of threads

[13]:
```
# Comparison on 10k users
sns.barplot(data=results_ten_k, x="factors", y="inference_speed", hue="framework",␣
↪palette=["steelblue", "crimson"]);
plt.xlabel('Factors')
plt.ylabel('Seconds')
plt.title("LightFM model inference speed for 10k users on Movielens 20m")
plt.show()
```



nbsphinx-code-borderwhite

[14]:
```
# Comparison on 140k users
sns.barplot(data=results_all, x="factors", y="inference_speed", hue="framework",␣
↪palette=["steelblue", "crimson"]);
plt.xlabel('Factors')
plt.ylabel('Seconds')
plt.title("LightFM model inference speed for all 140k users on Movielens 20m")
plt.show()
```



nbsphinx-code-borderwhite

Other benefits of RecTools wrapper

- No need for manual construction of sparse matrixes over interactions and users/items features
- No need for manual processing of retrived scores and ranking
- Model has the same interface with all other RecTools models and can be compared easily
- `filter_viewed` and `items_to_recommend` options

How is the speed optimized?

We are using vectorization of LightFM user/item feature factors and biases (which already provided x2 speed boost to the original library). For even faster inference we use implicit library optimized matrix_factorization top_k method over the constructed vectors.

3.9.7 Example of VisualApp for recommendations

Visual analysis is an important part of the models evaluation process. Let's see how to use interactive Jupyter widgets for visualization with RecTools

`visuals` extension for `rectools` is required to run this notebook. You can install it with `pip install rectools[visuals]`

Table of Contents

  - *Any html code from df values*
  - *Random users*
  - *Multiple models comparison*
  - *Saving and loading app*
  - *Item-to-item case*
- *Final app example*

[2]:
```
import pandas as pd

from rectools import Columns
from rectools.visuals import VisualApp, ItemToItemVisualApp
```

Prepare data

Imagine we have movie recommendations task. We trained a model and calculated recommendations for a set of users. Now we want to visualize results. Researcher usually wants to see both user's `interactions` history with the items, his `recommended items` from the model and also some additional `item data`.

[3]:
```
interactions = pd.DataFrame({
    "user_id": [10, 10, 20],
    "item_id": [1, 2, 4],
})
interactions
```

[3]:

	user_id	item_id
0	10	1
1	10	2
2	20	4

[4]:
```
reco = pd.DataFrame({
    "user_id": [10, 10, 20, 20],
    "item_id": [4, 0, 1, 2],
    "rank": [1, 2, 1, 2],
    "model": ["Random model"] * 4
})
reco
```

[4]:

	user_id	item_id	rank	model
0	10	4	1	Random model
1	10	0	2	Random model
2	20	1	1	Random model
3	20	2	2	Random model

[5]:
```
item_data = pd.DataFrame({
    "item_id": range(5),
    "name": ["Green book", "Meir from Easttown", "True detective", "Oppenheimer", "John␣
↪Wick"],
    "jenre": ["drama", "detective", "detective", "drama", "action"],
    "img_url": [
        'https://avatars.mds.yandex.net/get-kinopoisk-image/1900788/8e4206c9-fb99-4f43-
```

(continues on next page)

(continued from previous page)

```
↪9170-4586a5bc5b9b/3840x',
       'https://avatars.mds.yandex.net/get-kinopoisk-image/4774061/9c4e7a31-2b5a-4c8c-
↪abc3-bfd67b8778cb/576x',
       'https://avatars.mds.yandex.net/get-kinopoisk-image/1946459/f5d887aa-dee4-4158-
↪9713-61d41b04f94d/3840x',
       'https://avatars.mds.yandex.net/get-kinopoisk-image/6201401/9d064dee-0b29-4660-
↪881a-1e7a3f81b3da/3840x',
       'https://avatars.mds.yandex.net/get-kinopoisk-image/1946459/bed1d2f9-cf3a-46a2-
↪a6cd-cde4dc41ea43/3840x'
       ]
})
item_data
```

[5]:

	item_id	name	genre	\
0	0	Green book	drama	
1	1	Meir from Easttown	detective	
2	2	True detective	detective	
3	3	Oppenheimer	drama	
4	4	John Wick	action	

	img_url
0	https://avatars.mds.yandex.net/get-kinopoisk-i...
1	https://avatars.mds.yandex.net/get-kinopoisk-i...
2	https://avatars.mds.yandex.net/get-kinopoisk-i...
3	https://avatars.mds.yandex.net/get-kinopoisk-i...
4	https://avatars.mds.yandex.net/get-kinopoisk-i...

Simple visualization example

Now we wan't to visualize recommedations for users in a conveninent and interactive way using RecTools `VisualApp`.

[ ]:
```
app = VisualApp.construct(
    reco=reco,
    interactions=interactions,
    item_data=item_data,
    selected_users={"detective user": 10, "action user": 20},  # users that we want to␣
↪visualise
    formatters={"img_url": lambda x: f"<img src={x} width=100>"}  # process "img_url"␣
↪links to html code
)
```

If you run this notebook, you will get **interactive** widgets with active buttons to select users.
For offline presentation we keep **static** screenshots of the actual app.

Target:

detective user | action user

user_id: 10

Interactions

item_id	name	jenre	img_url
1	Meir from Easttown	detective	
2	True detective	detective	

Model:

Random model

Model name: Random model

Recommended

item_id	name	jenre	img_url	rank
3	Oppenheimer	drama		1
0	Green book	drama		2

Details on `VisualApp` features

1. How to process pd.DataFrame values into any desired html code (e.g. display actual images from links)

For this we need to prepare the `formatters` dict. Dataframe column names are keys and functions to generate html code are values. You can process any columns present in the app

[23]:
```
def image_html(url: str) -> str:
    return '<img src="' + url + '" style=max-height:150px;"/>'

formatters={"img_url":image_html}  # "img_url" here is a name of the column in `item_data`
```

2. How to add random users for visualization

Just pass `n_random_users` with the desired amount. Users will be selected from those that are present in recommendations

[24]:
```
n_random_users=2
```

3. How to pass multiple models for visualization

For this you can just add recommendations from another model to the same `reco` dataframe and write different model name in "model" column.
Or alternatively you can create a `TablesDict` with model names as keys and model recommendations dataframes as values. Just like in the example below:

[30]:
```
reco = {
    "model_1": pd.DataFrame({Columns.User: [10, 20], Columns.Item: [3, 4], Columns.Score:
↪ [0.99, 0.9]}),
    "model_2": pd.DataFrame({Columns.User: [10, 20], Columns.Item: [2, 0], Columns.Rank:␣
↪[1, 1]})
}
```

4. How to save `VisualApp` and easily access the same widgets later

Jupyter widgets disappear when notebook is closed. To recreate the same widgets use `save` and `load` methods of `VisualApp`.

[33]:
```
app = VisualApp.construct(
    reco=reco,
    interactions=interactions,
    item_data=item_data,
    n_random_users=n_random_users,
    formatters=formatters,
    auto_display=False  # prevent widgets from displaying
)
```

(continues on next page)

(continued from previous page)

```
app.save("sample_app")

# Next time just run:
# app = VisualApp.load("sample_app", formatters=formatters)
```

5. Hot to visualize item-to-item recommendations

For this case use `ItemToItemVisualApp`. Interface is almost the same, but there is no need for interactions now. Remember to specify target items in `Columns.TargetItem` column in recommendations

[ ]:
```
reco = {
    "model_1": pd.DataFrame({Columns.TargetItem: [1, 2], Columns.Item: [3, 4], Columns.
↪Score: [0.99, 0.9]}),
    "model_2": pd.DataFrame({Columns.TargetItem: [1, 2], Columns.Item: [2, 1], Columns.
↪Rank: [1, 1]})
}
```

Final app example

- item-to-item recommendations
- multiple models comparison
- random targets
- custom html code for column with images

[ ]:
```
app = ItemToItemVisualApp.construct(
    reco=reco,
    item_data=item_data,
    n_random_items=2,
    formatters=formatters,
)
```

If you run this notebook, you will get **interactive** widgets with active buttons to select items and models.
For offline presentation we keep **static** screenshots of the actual app.

Target:

random_1 random_2

target_item_id: 1

Interactions

item_id	name	jenre	img_url
1	Meir from Easttown	detective	

Model:

model_1 model_2

Model name: model_1

Recommended

item_id	name	jenre	img_url	score
3	Oppenheimer	drama		0.99

3.10 FAQ

1. **What kind of features should I use: Dense or Sparse?**
   It depends. In most cases you're better off using *SparseFeatures* because they are better suited to categorical features. Even if you have a feature with real numerical values you're often better off if you binarize or discretize it. But there are exceptions to this rule, e.g. ALS features.

2. **How do I calculate several metrics at once?**
   Use function *calc_metrics*. It allows to calculate a batch of metrics more efficiently. It's similar to *reports* from *sklearn*.

3. **What is the benefit of model wrappers?**
   They all have the same set of parameters allowing for easier usage. They also provide extension of existing functionality, such as allowing to filters to eliminate items that has already been seen, whitelist, features in ALS, I2I. Wrappers have unified interface of output that is easy to use as input to calculate metrics. They also allowed to speed up performance of some models.

4. **What is the benefit of using *Dataset*?**
   It's an easy-to-use wrapping of interactions, features and mapping between item and user ids in feature sets and those in interaction matrix.

5. **Why do I need to pass *Dataset* object as an argument to method *recommend*?**
   It conceals mapping between internal and external user and item ids. Additionally it allows to filter out

items that users have already seen. Some models, such as *LightFM* or *DSSM*, require to pass features.

6. **Should the same *Dataset* object be used for fitting of a model and for inference of recommendations?**
   It almost always has to be exactly the same *Dataset* object.

   One of possible exceptions is if during the fitting stage you use both viewing and purchase of an item as a positive event but you want exempt an item from being recommended only if it was purchased. In this case you should pass all interactions to train a model and only purchases to infer recommendations.

   Another exception is if a model requires to pass features to infer recommendations and values of those features have changed.

3.11 Support

If something went wrong and you can't find a way to fix it please contact us at Telegram Channel

PYTHON MODULE INDEX

r

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

P

R

S

T

U

V