
irspack

Release 0.1.0

Tomoki Ohtsuki

Jun 03, 2023

BASIC TUTORIALS

1	Installation	3
2	Indices and tables	125
	Index	127

irspack is a collection of recommender system algorithms for implicit feedback data.

Currently, in my opinion, there is no all-purpose algorithm for the recommendation tasks with implicit-feedback. So the key is to try out different algorithms, evaluate its performance against validation dataset, and optimize their performance by tuning hyperparameters. irspack is built to make this procedure easy for you.

Notable features include:

- [optuna](#)-backed, efficient hyperparameter optimization. In particular, [pruning](#) is used to speed-up the parameter search for several algorithms.
- Implementation of several parallelizable algorithms with [Pybind11](#) and [Eigen](#). Evaluation of recommenders' performance (which involves score-sorting and ranking metric computation) can be also done efficiently using these technologies.

INSTALLATION

If you have a standard Python environment on MacOS/Linux, you can install the library from PyPI using pip:

```
pip install irspack
```

The binaries on PyPI have been built to use AVX instruction. If you want to use AVX2/AVX512 or your environment does not support AVX (e.g. Rosetta2 on Apple Silicon), install it from source by e.g., :

```
CFLAGS="-march=native" pip install git+https://github.com/tohtsky/irspack.git
```

1.1 Train our first movie recommender

In this tutorial, we build our first recommender system using a simple algorithm called **P3alpha**.

We will learn

- How to represent the implicit feedback dataset as a sparse matrix.
- How to fit **irspack**'s models using the sparse matrix representation.
- How to make a recommendation using our API.

```
[1]: import numpy as np
import scipy.sparse as sps

from irspack.dataset import MovieLens1MDataManager
from irspack import P3alphaRecommender
```

1.1.1 Read Movielens 1M dataset

We first load the **Movielens1M** dataset. For the first time, you will be asked to allow downloading the dataset.

```
[2]: loader = MovieLens1MDataManager()

df = loader.read_interaction()
df.head()
```

```
[2]:   userId  movieId  rating      timestamp
0      1      1193       5 2000-12-31 22:12:40
1      1       661       3 2000-12-31 22:35:09
2      1       914       3 2000-12-31 22:32:48
```

(continues on next page)

(continued from previous page)

3	1	3408	4	2000-12-31 22:04:35
4	1	2355	5	2001-01-06 23:38:11

df stores the users' watch event history.

Although the rating information is available in this case, we will not be using this column. What matters to implicit feedback based recommender system is “which user interacted with which item (movie)”.

By loader we can also read the dataframe for the movie meta data:

```
[3]: movies = loader.read_item_info()
     movies.head()
```

```
[3]:
```

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


```

release_year
movieId
1      1995
2      1995
3      1995
4      1995
5      1995

```

1.1.2 Represent your data as a sparse matrix

We represent the data as a sparse matrix X , whose element X_{ui} is given by

$$X_{ui} = \begin{cases} 1 & \text{if the user } u \text{ has watched the item (movie) } i \\ 0 & \text{otherwise} \end{cases}$$

For this purpose, we use `np.unique` function with `return_inverse=True`. This will return a tuple that consists of

1. The list of unique user/movie ids appearing in the original user/movie id array
2. How the original user/movie id array elements are mapped to the array 1.

So if we do

```
[4]: unique_user_ids, user_index = np.unique(df.userId, return_inverse=True)
     unique_movie_ids, movie_index = np.unique(df.movieId, return_inverse=True)
```

then `unique_user_ids[user_index]` and `unique_movie_ids[movie_index]` is equal to the original array:

```
[5]: assert np.all( unique_user_ids[user_index] == df.userId.values )
     assert np.all( unique_movie_ids[movie_index] == df.movieId.values )
```

Thus, we can think of `user_index` and `movie_index` as representing the row and column positions of non-zero elements, respectively.

Now X can be constructed as `scipy's` sparse `csr matrix` as follows.


```
[6]: X = sps.csr_matrix(
    (
        np.ones(df.shape[0]), # values of non-zero elements
        (
            user_index, # rows of non-zero elements
            movie_index # cols of non-zero elements
        )
    )
)

X
```

```
[6]: <6040x3706 sparse matrix of type '<class 'numpy.float64'>'
      with 1000209 stored elements in Compressed Sparse Row format>
```

We encounter this pattern so often, so there is `df_to_sparse` function in `irspack`:

```
[7]: from irspack import df_to_sparse
X_, unique_user_ids_, unique_item_ids_ = df_to_sparse(df, 'userId', 'movieId')

# X_ is identical to X.
assert (X_ - X).getnnz() == 0
```

1.1.3 Fit the recommender.

We fit `P3alphaRecommender` against `X`.

```
[8]: recommender = P3alphaRecommender(X)
recommender.learn()

[8]: <irspack.recommenders.p3.P3alphaRecommender at 0x7f61e7e679d0>
```

1.1.4 Check the recommender's output

Suppose there is a new user who has just watched “Toy Story”. Let us see what would be the recommended for this user.

We first represent the user's watch profile as another sparse matrix (which contains a single non-zero element).

```
[9]: movie_id_vs_movie_index = { mid: i for i, mid in enumerate(unique_movie_ids)}

toystory_id = 1
toystory_watcher_matrix = sps.csr_matrix(
    ([1], ([0], [movie_id_vs_movie_index[toystory_id]])),
    shape=(1, len(unique_movie_ids)) # this time shape parameter is required
)

movies.loc[toystory_id]
```

```
[9]: title          Toy Story (1995)
genres      Animation|Children's|Comedy
release_year          1995
Name: 1, dtype: object
```

Since this user is new (previously unseen) to the recommender, we use `get_score_cold_user_remove_seen` method.

`remove_seen` means that we mask the scores for the items that user had watched already (in this case, Toy Story) so that such items would not be recommended again.

As you can see, the score corresponding to “Toy Story” has $-\infty$ score.

```
[10]: score = recommender.get_score_cold_user_remove_seen(
      toystory_watcher_matrix
    )

      # Id 1 (index 0) is masked (have -infinity score)
      score
[10]: array([[          -inf,  8.18606963e-04,  4.30083199e-04, ...,
            4.30589311e-05,  1.09994485e-05,  2.71571993e-04]])
```

To get the recommendation, we argsort wthe score by descending order and convert “movie index” (which starts from 0) to “movie id”.

```
[11]: recommended_movie_index = score[0].argsort()[::-1][:10]
      recommended_movie_ids = unique_movie_ids[recommended_movie_index]

      # Top-10 recommendations
      recommended_movie_ids
[11]: array([2858, 1265, 2396, 3114,  260, 1210, 1196, 1270, 2028,  34])
```

And here are the titles of the recommendations.

```
[12]: movies.reindex(recommended_movie_ids)
[12]:
```

movieId	title \
2858	American Beauty (1999)
1265	Groundhog Day (1993)
2396	Shakespeare in Love (1998)
3114	Toy Story 2 (1999)
260	Star Wars: Episode IV - A New Hope (1977)
1210	Star Wars: Episode VI - Return of the Jedi (1983)
1196	Star Wars: Episode V - The Empire Strikes Back...
1270	Back to the Future (1985)
2028	Saving Private Ryan (1998)
34	Babe (1995)

movieId	genres	release_year
2858	Comedy Drama	1999
1265	Comedy Romance	1993
2396	Comedy Romance	1998
3114	Animation Children's Comedy	1999
260	Action Adventure Fantasy Sci-Fi	1977
1210	Action Adventure Romance Sci-Fi War	1983
1196	Action Adventure Drama Sci-Fi War	1980
1270	Comedy Sci-Fi	1985

(continues on next page)

(continued from previous page)

2028	Action Drama War	1998
34	Children's Comedy Drama	1995

The above pattern - mapping item IDs to indexes, creating sparse matrices, and reverting indexes of recommended items to item IDs - is a quite common one, and we have also created a convenient class that does the item index/ID mapping:

```
[13]: from irspack.utils.id_mapping import ItemIDMapper

id_mapper = ItemIDMapper(
    item_ids=unique_movie_ids
)
id_and_scores = id_mapper.recommend_for_new_user(
    recommender,
    [toystory_id], cutoff = 10
)
movies.reindex(
    [ item_id for item_id, score in id_and_scores ]
)
```

```
[13]:
```

	title \
movieId	
2858	American Beauty (1999)
1265	Groundhog Day (1993)
2396	Shakespeare in Love (1998)
3114	Toy Story 2 (1999)
260	Star Wars: Episode IV - A New Hope (1977)
1210	Star Wars: Episode VI - Return of the Jedi (1983)
1196	Star Wars: Episode V - The Empire Strikes Back...
1270	Back to the Future (1985)
2028	Saving Private Ryan (1998)
34	Babe (1995)

	genres	release_year
movieId		
2858	Comedy Drama	1999
1265	Comedy Romance	1993
2396	Comedy Romance	1998
3114	Animation Children's Comedy	1999
260	Action Adventure Fantasy Sci-Fi	1977
1210	Action Adventure Romance Sci-Fi War	1983
1196	Action Adventure Drama Sci-Fi War	1980
1270	Comedy Sci-Fi	1985
2028	Action Drama War	1998
34	Children's Comedy Drama	1995

While the above result might make sense, this is not an optimal result. To get better results, we have to tune the recommender's hyper parameters against some accuracy metric measured on a validation set.

In the next tutorial, we will see how to define the hold-out and validation score.

1.2 Evaluation of recommender systems

In this tutorial, we explain how to evaluate recommender systems with implicit feedback by holding out method.

```
[1]: from irspack.dataset import MovieLens1MDataManager
from irspack import (
    P3alphaRecommender, TopPopRecommender,
    rowwise_train_test_split, Evaluator,
    df_to_sparse
)
```

1.2.1 Read the ML1M dataset again.

As in the previous tutorial, we load the rating dataset and construct a sparse matrix.

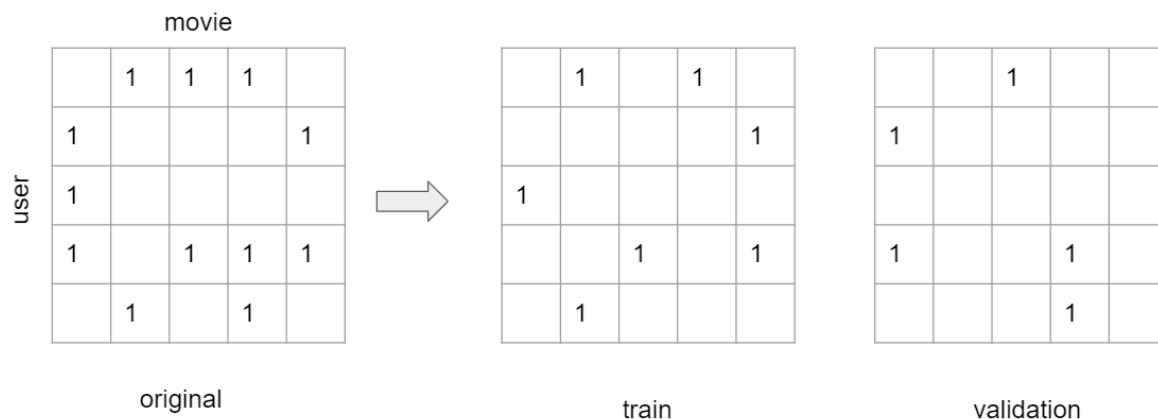
```
[2]: loader = MovieLens1MDataManager()

df = loader.read_interaction()

X, unique_user_ids, unique_movie_ids = df_to_sparse(
    df, 'userId', 'movieId'
)
```

1.2.2 Split scheme 1. Hold-out for all users.

To evaluate the performance of a recommender system trained with implicit feedback, the standard method is to hide some subset of the known user-item interactions as a validation set and see how the recommender ranks these hidden groundtruths:



We have prepared a fast implementation of such a split (with random selection of these subset) in `rowwise_train_test_split` function:

```
[3]: X_train, X_valid = rowwise_train_test_split(X, test_ratio=0.2, random_state=0)

assert X_train.shape == X_valid.shape
```

They sum back to the original matrix:

```
[4]: X - (X_train + X_valid) # 0 stored elements

[4]: <6040x3706 sparse matrix of type '<class 'numpy.float64'>'
      with 0 stored elements in Compressed Sparse Row format>
```

There is no overlap of non-zero elements:

```
[5]: X_train.multiply(X_valid) # Element-wise multiplication yields 0 stored elements

[5]: <6040x3706 sparse matrix of type '<class 'numpy.float64'>'
      with 0 stored elements in Compressed Sparse Row format>
```

This scheme however has a problem regarding the performance because we have to compute the recommendation score for all the users. So in the next tutorial, we will be using a sub-sampled version of this splitting.

1.2.3 Obtain the evaluation metric

Now we define the Evaluator object, which will measure the performance of various recommender systems based on `X_valid` (the meaning of `offset=0` will be clarified in the next tutorial).

```
[6]: evaluator = Evaluator(X_valid, offset=0)
```

We fit `P3alphaRecommender` using `X_train`, and compute its accuracy metrics against `X_valid` using `evaluator`.

Internally, `evaluator` calls the recommender's `get_score_remove_seen` method, sorts the score to obtain the rank, and reconciles it with the stored validation interactions.

```
[7]: recommender = P3alphaRecommender(X_train, top_k=100)
      recommender.learn()

      evaluator.get_scores(recommender, cutoffs=[5, 10])

[7]: OrderedDict([('hit@5', 0.7988410596026491),
                  ('recall@5', 0.09333713653053713),
                  ('ndcg@5', 0.396153990817333),
                  ('map@5', 0.06632471989643292),
                  ('precision@5', 0.37317880794701996),
                  ('gini_index@5', 0.9738255665593293),
                  ('entropy@5', 4.777123828102488),
                  ('appeared_item@5', 536.0),
                  ('hit@10', 0.8925496688741722),
                  ('recall@10', 0.15053594583416965),
                  ('ndcg@10', 0.3662089065311077),
                  ('map@10', 0.08982972949880254),
                  ('precision@10', 0.32049668874172194),
                  ('gini_index@10', 0.961654047669253),
                  ('entropy@10', 5.1898345912683315),
                  ('appeared_item@10', 764.0)])
```

1.2.4 Comparison with a simple baseline

Now that we have a qualitative way to measure the recommenders' performance, we can compare the performance of different algorithms.

As a simple baseline, we fit `TopPopRecommender`, which recommends items with descending order of the popularity in the train set, regardless of the users' watch event history. (But note that already-seen items by a user will not be commended again).

```
[8]: toppop_recommender = TopPopRecommender(X_train)
    toppop_recommender.learn()

evaluator.get_scores(toppop_recommender, cutoffs=[5, 10])

[8]: OrderedDict([('hit@5', 0.5473509933774835),
                  ('recall@5', 0.04097159018092859),
                  ('ndcg@5', 0.21914854330480912),
                  ('map@5', 0.025239375245273265),
                  ('precision@5', 0.20884105960264907),
                  ('gini_index@5', 0.9972226173414867),
                  ('entropy@5', 2.608344593727912),
                  ('appeared_item@5', 42.0),
                  ('hit@10', 0.6637417218543047),
                  ('recall@10', 0.0667908851617647),
                  ('ndcg@10', 0.19939297384808613),
                  ('map@10', 0.033298013667913656),
                  ('precision@10', 0.17811258278145692),
                  ('gini_index@10', 0.9950046639957398),
                  ('entropy@10', 3.1812807131889786),
                  ('appeared_item@10', 69.0)])
```

So we see that `P3alphaRecommender` actually exhibits better accuracy scores compared to rather trivial `TopPopRecommender`.

In the next tutorial, we will optimize the recommender's performance using the hold-out method.

1.3 Hyperparameter Optimization

In this tutorial, we first demonstrate how `P3alphaRecommender`'s performance can be optimized by `optuna`-backed `tune` function.

Then, by further splitting the ground-truth interaction into tran, validation and test ones, we compare several recommenders' performance optimized on the validation set and measured on the test set.

```
[1]: from IPython.display import clear_output, display
    import numpy as np
    import scipy.sparse as sps
    from sklearn.model_selection import train_test_split

    from irspack.dataset import MovieLens1MDataManager
    from irspack import (
        P3alphaRecommender, rowwise_train_test_split, Evaluator,
        df_to_sparse
    )
```

1.3.1 Read the ML1M dataset again.

We again prepare the sparse matrix X .

```
[2]: loader = MovieLens1MDataManager()

df = loader.read_interaction()

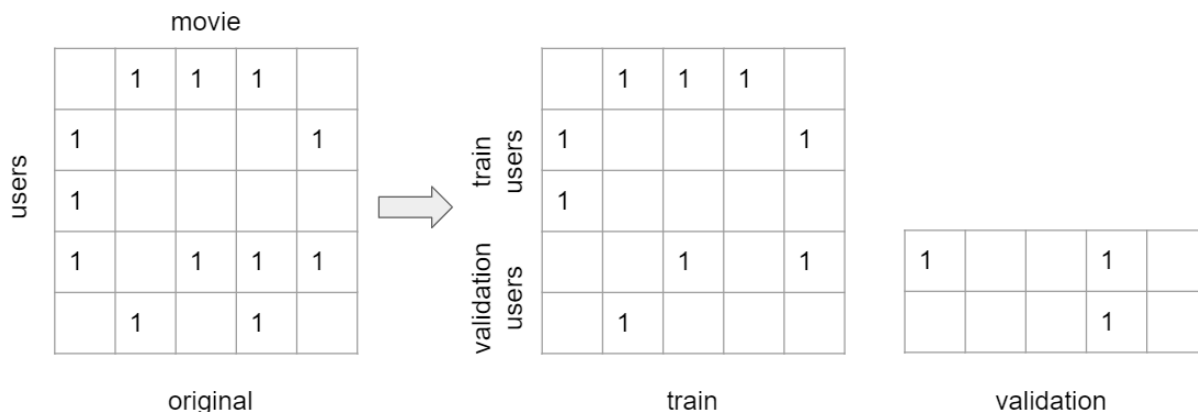
movies = loader.read_item_info()
movies.head()

X, unique_user_ids, unique_movie_ids = df_to_sparse(
    df, 'userId', 'movieId'
)
```

1.3.2 Split scheme 2. Hold-out for partial users.

To perform the hyperparameter optimization, we have to repeatedly measure the accuracy metrics on the validation set. As mentioned in the previous tutorial, doing this for all users is time-consuming (often heavier than the recommender's learning process), so we truncate this subset as follows:

1. First split **users** into “train”, “validation” (and “test”) ones.
2. For train users, feed all their interactions into the recommender. For validation (test) users, hold-out part of their interaction for the validation (“prediction” part), and feed the rest (“learning” part) into the recommender.
3. After the fit, ask the recommender to output the score only for validation (test) users, and see how it ranks these held-out interactions for the validation (test) users.



Although we have prepared another function to do this procedure, let us first do this manually.

```
[3]: # Split users into train and validation users.

X_train_user, X_valid_user = train_test_split(X, test_size=.4, random_state=0)

# Split the validation users' interaction into learning 50% and prediction 50%.

X_valid_learn, X_valid_predict = rowwise_train_test_split(
```

(continues on next page)

(continued from previous page)

```
X_valid_user, test_ratio=.5, random_state=0
)
```

1.3.3 Define the evaluator and optimize the validation metric

As illustrated above, we will use

- Train users' all interactions (`X_train_user`)
- Validation users' 50% interaction (`X_valid_learn`)

as the recommender's training resource, and validation users' rest interaction (`X_valid_predict`) as the held-out ground truth:

```
[4]: X_train_val_learn = sps.vstack([X_train_user, X_valid_learn])
      evaluator = Evaluator(X_valid_predict, offset=X_train_user.shape[0], target_metric='ndcg',
      ↪ cutoff=20)
```

The offset parameter specifies where the validation user block begins (where the train user block ends).

Now to start the optimization.

```
[5]: best_params, validation_results = P3alphaRecommender.tune(X_train_val_learn, evaluator,
      ↪ random_seed=0, n_trials=20)
      clear_output() # output is a bit lengthy
```

The best `ndcg@20` value is

```
[6]: validation_results['ndcg@20'].max()
[6]: 0.5159628863136182
```

which has been obtained by using these hyper parameters:

```
[7]: best_params
[7]: {'top_k': 217, 'normalize_weight': True}
```

Meanwhile, the default argument of `P3alphaRecommnder` (which has been used so far) attains `ndcg@20` = 0.4084. So this is indeed a significant improvement:

```
[8]: rec_default = P3alphaRecommender(X_train_val_learn).learn()
      evaluator.get_score(rec_default)['ndcg']
[8]: 0.4084060191998281
```


1.3.4 Check the recommender's output again

Let us check how our recommender has evolved from the first tutorial. We consider the same setting (a new user has watched “Toy Story”), but fit the recommender using the obtained parameters.

```
[9]: rec_tuned = P3alphaRecommender(X, **best_params).learn()
```

```
from irspack import ItemIDMapper
id_mapper = ItemIDMapper(unique_movie_ids)
```

```
[10]: toystory_id = 1
recommended_id_and_score = id_mapper.recommend_for_new_user(
    rec_tuned, user_profile=[toystory_id], cutoff=10
)

# Top-10 recommendations
movies.reindex([movie_id for movie_id, score in recommended_id_and_score])
```

```
[10]:
```

	movieId	title	genres \
	1265	Groundhog Day (1993)	Comedy Romance
	2396	Shakespeare in Love (1998)	Comedy Romance
	3114	Toy Story 2 (1999)	Animation Children's Comedy
	1270	Back to the Future (1985)	Comedy Sci-Fi
	2028	Saving Private Ryan (1998)	Action Drama War
	34	Babe (1995)	Children's Comedy Drama
	2571	Matrix, The (1999)	Action Sci-Fi Thriller
	356	Forrest Gump (1994)	Comedy Romance War
	2355	Bug's Life, A (1998)	Animation Children's Comedy
	1197	Princess Bride, The (1987)	Action Adventure Comedy Romance

	movieId	release_year
	1265	1993
	2396	1998
	3114	1999
	1270	1985
	2028	1998
	34	1995
	2571	1999
	356	1994
	2355	1998
	1197	1987

Note how drastically the recommended contents have changed (increased significance of genre “Children’s” and disappearance of “Star Wars” series, etc...).

1.3.5 A train/validation/test split example

To rigorously compare the performance of various recommender algorithms, we should measure the final score against the **test** dataset, not the validation set, and it is straightforward now.

To begin with, we have prepared a function called `split_dataframe_partial_user_holdout` which splits the users in the original dataframe into train/validation/test users, holding out partial interaction for validation/test user:

```
[11]: from irspack.split import split_dataframe_partial_user_holdout

dataset, item_ids = split_dataframe_partial_user_holdout(
    df, 'userId', 'movieId', val_user_ratio=.3, test_user_ratio=.3,
    heldout_ratio_val=.5, heldout_ratio_test=.5
)

dataset
```

```
[11]: {'train': <irspack.split.userwise.UserTrainTestInteractionPair at 0x7ff61f483430>,
      'val': <irspack.split.userwise.UserTrainTestInteractionPair at 0x7ff61f4831f0>,
      'test': <irspack.split.userwise.UserTrainTestInteractionPair at 0x7ff61f482bc0>}
```

As you can see, the returned dataset is a dictionary which stores train/validation/test-users' interactions as an instance of `UserTrainTestInteractionPair`.

```
[12]: train_users = dataset['train']
      val_users = dataset['val']
      test_users = dataset['test']

      # Concatenate train/validation users into one.
      train_and_val_users = train_users.concat(val_users)
```

```
[13]: val_users.X_train
```

```
[13]: <1812x3706 sparse matrix of type '<class 'numpy.float64'>'
      with 152333 stored elements in Compressed Sparse Row format>
```

```
[14]: val_users.X_test
```

```
[14]: <1812x3706 sparse matrix of type '<class 'numpy.float64'>'
      with 151435 stored elements in Compressed Sparse Row format>
```

```
[15]: val_users.X_all # which equals val_users.X_train + val_users.X_test
```

```
[15]: <1812x3706 sparse matrix of type '<class 'numpy.float64'>'
      with 303768 stored elements in Compressed Sparse Row format>
```

```
[16]: # For train users, there is no "test" interaction held out.
      train_users.X_test
```

```
[16]: <2416x3706 sparse matrix of type '<class 'numpy.float64'>'
      with 0 stored elements in Compressed Sparse Row format>
```

For each recommender algorithm (here P3alpha, RP3beta, IALS and DenseSLIM), we perform:

1. Hyperparameter optimization. During this phase, we will be using train users' all interaction and validation users' train interaction as the source of learning, and validation users' test interaction as the held-out ground truth.

2. Evaluation. During this phase, we will include train/validation users' all interactions as well as test users' train interaction as the source of learning, and fit the model using the parameters obtained in the optimization phase. Then we measure the recommender's performance against **test** users' test interaction.

```
[17]: from typing import Type
      from irspack import DenseSLIMRecommender, RP3betaRecommender, IALSRecommender, \
      ↪BaseRecommender

[18]: val_evaluator = Evaluator(
      val_users.X_test,
      offset=train_users.n_users,
      cutoff=20, target_metric="ndcg"
    )
    test_evaluator = Evaluator(
      test_users.X_test,
      offset=train_and_val_users.n_users
    )
    test_results = []
    recommender_name_vs_best_parameter = {}
    recommender_class: Type[BaseRecommender]
    for recommender_class in [IALSRecommender, DenseSLIMRecommender, P3alphaRecommender, \
    ↪RP3betaRecommender]:
        print(f'Start tuning {recommender_class.__name__}.')
        best_params, validation_results_df = recommender_class.tune(
            sps.vstack([train_users.X_all, val_users.X_train]),
            val_evaluator, n_trials=40, random_seed=0
        )
        recommender = recommender_class(
            sps.vstack([train_and_val_users.X_all, test_users.X_train]),
            **best_params
        ).learn()
        recommender_name_vs_best_parameter[recommender_class.__name__] = best_params

        test_score = dict(
            algorithm=recommender_class.__name__,
            **test_evaluator.get_scores(recommender, cutoffs=[20])
        )
        test_results.append(test_score)
    clear_output()
```

As you can see below, iALS and DenseSLIM outperforms others in terms of accuracy measures (recall, ndcg, map). iALS performed well regarding the diversity scores (entropy, gini-index, appeared_item), too.

```
[19]: import pandas as pd
      pd.DataFrame(test_results)

[19]:
```

	algorithm	hit@20	recall@20	ndcg@20	map@20	\
0	IALSRecommender	0.996137	0.208540	0.576164	0.135950	
1	DenseSLIMRecommender	0.995033	0.207463	0.572965	0.135436	
2	P3alphaRecommender	0.993377	0.182982	0.526259	0.114564	
3	RP3betaRecommender	0.995033	0.188152	0.537070	0.119353	
	precision@20	gini_index@20	entropy@20	appeared_item@20		

(continues on next page)

(continued from previous page)

0	0.528201	0.915915	5.989211	1108.0
1	0.525055	0.926926	5.859120	1018.0
2	0.477152	0.962736	5.174319	690.0
3	0.486010	0.957136	5.297020	846.0

Let's ask each recommender, "What would you recommend to a user who has just seen "Toy Story"?"

IALS, DenseSLIM, RP3beta rank "Toy Story2" at the top of recommendationlist, which seems appropriate.

```
[20]: for recommender_class in [IALSRecommender, DenseSLIMRecommender, RP3betaRecommender,
    ↪ P3alphaRecommender]:
    rec_tuned = recommender_class(X, **recommender_name_vs_best_parameter[recommender_
    ↪ class.__name__]).learn()

    toystory_id = 1
    recommended_id_and_score = id_mapper.recommend_for_new_user(
        rec_tuned, user_profile=[toystory_id], cutoff=10
    )
    print(f"{recommender_class.__name__}'s result:")
    # Top-10 recommendations
    display(movies.reindex([movie_id for movie_id, score in recommended_id_and_score]))
```

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

IALSRecommender's result:

movieId	title	genres \
3114	Toy Story 2 (1999)	Animation Children's Comedy
34	Babe (1995)	Children's Comedy Drama
2355	Bug's Life, A (1998)	Animation Children's Comedy
1265	Groundhog Day (1993)	Comedy Romance
588	Aladdin (1992)	Animation Children's Comedy Musical
2396	Shakespeare in Love (1998)	Comedy Romance
2321	Pleasantville (1998)	Comedy
356	Forrest Gump (1994)	Comedy Romance War
1148	Wrong Trousers, The (1993)	Animation Comedy
595	Beauty and the Beast (1991)	Animation Children's Musical

release_year

movieId	
3114	1999
34	1995
2355	1998
1265	1993
588	1992
2396	1998
2321	1998
356	1994
1148	1993
595	1991

DenseSLIMRecommender's result:

	title \	
movieId		
3114	Toy Story 2 (1999)	
2355	Bug's Life, A (1998)	
34	Babe (1995)	
588	Aladdin (1992)	
1265	Groundhog Day (1993)	
2396	Shakespeare in Love (1998)	
356	Forrest Gump (1994)	
1148	Wrong Trousers, The (1993)	
1641	Full Monty, The (1997)	
1923	There's Something About Mary (1998)	
	genres	release_year
movieId		
3114	Animation Children's Comedy	1999
2355	Animation Children's Comedy	1998
34	Children's Comedy Drama	1995
588	Animation Children's Comedy Musical	1992
1265	Comedy Romance	1993
2396	Comedy Romance	1998
356	Comedy Romance War	1994
1148	Animation Comedy	1993
1641	Comedy	1997
1923	Comedy	1998
RP3betaRecommender's result:		
	title \	
movieId		
3114	Toy Story 2 (1999)	
1265	Groundhog Day (1993)	
2396	Shakespeare in Love (1998)	
34	Babe (1995)	
2355	Bug's Life, A (1998)	
1270	Back to the Future (1985)	
260	Star Wars: Episode IV - A New Hope (1977)	
2028	Saving Private Ryan (1998)	
356	Forrest Gump (1994)	
1210	Star Wars: Episode VI - Return of the Jedi (1983)	
	genres	release_year
movieId		
3114	Animation Children's Comedy	1999
1265	Comedy Romance	1993
2396	Comedy Romance	1998
34	Children's Comedy Drama	1995
2355	Animation Children's Comedy	1998
1270	Comedy Sci-Fi	1985
260	Action Adventure Fantasy Sci-Fi	1977
2028	Action Drama War	1998
356	Comedy Romance War	1994
1210	Action Adventure Romance Sci-Fi War	1983

P3alphaRecommender's result:

movieId	title	genres \
1265	Groundhog Day (1993)	Comedy Romance
2396	Shakespeare in Love (1998)	Comedy Romance
3114	Toy Story 2 (1999)	Animation Children's Comedy
1270	Back to the Future (1985)	Comedy Sci-Fi
2028	Saving Private Ryan (1998)	Action Drama War
34	Babe (1995)	Children's Comedy Drama
356	Forrest Gump (1994)	Comedy Romance War
2355	Bug's Life, A (1998)	Animation Children's Comedy
1197	Princess Bride, The (1987)	Action Adventure Comedy Romance
588	Aladdin (1992)	Animation Children's Comedy Musical

movieId	release_year
1265	1993
2396	1998
3114	1999
1270	1985
2028	1998
34	1995
356	1994
2355	1998
1197	1987
588	1992

1.4 API References

1.4.1 Evaluators

<i>Evaluator</i>	Evaluates recommenders' performance against validation set.
<i>EvaluatorWithColdUser</i>	Evaluates recommenders' performance against cold (unseen) users.

irspack.evaluation.Evaluator

```
class irspack.evaluation.Evaluator(ground_truth, offset=0, cutoff=10, target_metric='ndcg',
                                   recommendable_items=None, per_user_recommended_items=None,
                                   masked_interactions=None, n_threads=None,
                                   recall_with_cutoff=False, mb_size=128)
```

Bases: object

Evaluates recommenders' performance against validation set.

Parameters

- **ground_truth** (*Union[scipy.sparse.csr_matrix, scipy.sparse.csc_matrix]*)
– The ground-truth.

- **offset** (*int*) – Where the validation target user block begins. Often the validation set is defined for a subset of users. When offset is not 0, we assume that the users with validation ground truth corresponds to $X_{\text{train}}[\text{offset}]:]$ where X_{train} is the matrix feeded into the recommender class. Defaults to 0.
- **cutoff** (*int*, *optional*) – Controls the default number of recommendation. When the evaluator is used for parameter tuning, this cutoff value will be used. Defaults to 10.
- **target_metric** (*str*, *optional*) – Specifies the target metric when this evaluator is used for parameter tuning. Defaults to “ndcg”.
- **recommendable_items** (*Optional[List[int]]*, *optional*) – Global recommendable items. Defaults to None. If this parameter is not None, evaluator will be concentrating on the recommender’s score output for these recommendable_items, and compute the ranking performance within this subset.
- **per_user_recommendable_items** (*Union[None, List[List[int]]*, *csr_matrix*, *csc_matrix*) – Similar to *recommendable_items*, but this time the recommendable items can vary among users. If a sparse matrix is given, its nonzero indices are regarded as the list of recommendable items. Defaults to *None*.
- **masked_interactions** (*Optional[csr_matrix]*) – If set, this matrix masks the score output of recommender model where it is non-zero. If none, the mask will be the training matrix itself owned by the recommender.
- **n_threads** (*Optional[int]*) – Specifies the Number of threads to sort scores and compute the evaluation metrics. If *None*, the environment variable “IRSPACK_NUM_THREADS_DEFAULT” will be looked up, and if the variable is not set, it will be set to `os.cpu_count()`. Defaults to *None*.
- **recall_with_cutoff** (*bool*, *optional*) – This affects the definition of recall. If True, for each user, recall will be computed as

$$\frac{N_{\text{hit}}}{\min(\text{cutoff}, N_{\text{ground truth}})}$$

If False, this will be

$$\frac{N_{\text{hit}}}{N_{\text{ground truth}}}$$

- **mb_size** (*int*, *optional*) – The rows of chunked user score. Defaults to 1024.

__init__ (*ground_truth*, *offset=0*, *cutoff=10*, *target_metric='ndcg'*, *recommendable_items=None*, *per_user_recommendable_items=None*, *masked_interactions=None*, *n_threads=None*, *recall_with_cutoff=False*, *mb_size=128*)

Parameters

- **ground_truth** (*Union[csr_matrix, csc_matrix]*) –
- **offset** (*int*) –
- **cutoff** (*int*) –
- **target_metric** (*str*) –
- **recommendable_items** (*Optional[List[int]]*) –
- **per_user_recommendable_items** (*Union[None, List[List[int]]*, *csr_matrix*, *csc_matrix*) –
- **masked_interactions** (*Optional[Union[csr_matrix, csc_matrix]]*) –

- **n_threads** (*Optional[int]*) –
- **recall_with_cutoff** (*bool*) –
- **mb_size** (*int*) –

Return type

None

Methods

<code>__init__</code> (ground_truth[, offset, cutoff, ...])	
<code>get_score</code> (model)	Compute the score with the cutoff being <code>self.cutoff</code> .
<code>get_scores</code> (model, cutoffs)	Compute the score with the specified cutoffs.
<code>get_target_score</code> (model)	Compute the optimization target score (<code>self.target_metric</code>) with the cutoff being <code>self.cutoff</code> .

Attributes

<code>n_users</code>
<code>n_items</code>
<code>masked_interactions</code>

get_score(model)Compute the score with the cutoff being `self.cutoff`.**Parameters****model** ([BaseRecommender](#)) – The evaluated recommender.**Returns**

metric values.

Return type*Dict*[str, float]**get_scores(model, cutoffs)**

Compute the score with the specified cutoffs.

Parameters

- **model** ([BaseRecommender](#)) – The evaluated recommender.
- **cutoffs** (*List[int]*) – for each value in cutoff, the class computes the metric values.

ReturnsThe Resulting metric values. This time, the result will look like `{"ndcg@20": 0.35, "map@20": 0.2, ...}`.**Return type***Dict*[str, float]

get_target_score(model)

Compute the optimization target score (self.target_metric) with the cutoff being self.cutoff.

Parameters

model ([BaseRecommender](#)) – The evaluated model.

Returns

The metric value.

Return type

float

irspack.evaluation.EvaluatorWithColdUser

```
class irspack.evaluation.EvaluatorWithColdUser(input_interaction, ground_truth, cutoff=10,
                                              target_metric='ndcg', recommendable_items=None,
                                              per_user_recommended_items=None,
                                              masked_interactions=None, n_threads=None,
                                              recall_with_cutoff=False, mb_size=1024)
```

Bases: [Evaluator](#)

Evaluates recommenders' performance against cold (unseen) users.

Parameters

- **input_interaction** (*Union[[scipy.sparse.csr_matrix](#), [scipy.sparse.csc_matrix](#)]*) – The cold-users' known interaction with the items.
- **ground_truth** (*Union[[scipy.sparse.csr_matrix](#), [scipy.sparse.csc_matrix](#)]*) – The held-out ground-truth.
- **offset** (*int*) – Where the validation target user block begins. Often the validation set is defined for a subset of users. When offset is not 0, we assume that the users with validation ground truth corresponds to X_train[offset:] where X_train is the matrix feeded into the recommender class.
- **cutoff** (*int, optional*) – Controls the number of recommendation. Defaults to 10.
- **target_metric** (*str, optional*) – Optimization target metric. Defaults to "ndcg".
- **recommendable_items** (*Optional[List[int]], optional*) – Global recommendable items. Defaults to None. If this parameter is not None, evaluator will be concentrating on the recommender's score output for these recommendable_items, and compute the ranking performance within this subset.
- **per_user_recommended_items** (*Optional[List[List[int]]], optional*) – Similar to *recommendable_items*, but this time the recommendable items can vary among users. Defaults to None.
- **masked_interactions** (*Optional[Union[[scipy.sparse.csr_matrix](#), [scipy.sparse.csc_matrix](#)]], optional*) – If set, this matrix masks the score output of recommender model where it is non-zero. If none, the mask will be the training matrix (input_interaction) it self.
- **n_threads** (*int, optional*) – Specifies the Number of threads to sort scores and compute the evaluation metrics. If None, the environment variable "IRSPACK_NUM_THREADS_DEFAULT" will be looked up, and if the variable is not set, it will be set to `os.cpu_count()`. Defaults to None.

- **recall_with_cutoff** (*bool*, *optional*) – This affects the definition of recall. If True, for each user, recall will be evaluated by

$$\frac{N_{\text{hit}}}{\min(\text{cutoff}, N_{\text{ground truth}})}$$

If False, this will be

$$\frac{N_{\text{hit}}}{N_{\text{ground truth}}}$$

- **mb_size** (*int*, *optional*) – The rows of chunked user score. Defaults to 1024.

__init__ (*input_interaction*, *ground_truth*, *cutoff=10*, *target_metric='ndcg'*, *recommendable_items=None*, *per_user_recommendable_items=None*, *masked_interactions=None*, *n_threads=None*, *recall_with_cutoff=False*, *mb_size=1024*)

Parameters

- **input_interaction** (*Union[csr_matrix, csc_matrix]*) –
- **ground_truth** (*Union[csr_matrix, csc_matrix]*) –
- **cutoff** (*int*) –
- **target_metric** (*str*) –
- **recommendable_items** (*Optional[List[int]]*) –
- **per_user_recommendable_items** (*Union[None, List[List[int]], csr_matrix, csc_matrix]*) –
- **masked_interactions** (*Optional[Union[csr_matrix, csc_matrix]]*) –
- **n_threads** (*Optional[int]*) –
- **recall_with_cutoff** (*bool*) –
- **mb_size** (*int*) –

Methods

<code>__init__(input_interaction, ground_truth[, ...])</code>	
<code>get_score(model)</code>	Compute the score with the cutoff being <code>self.cutoff</code> .
<code>get_scores(model, cutoffs)</code>	Compute the score with the specified cutoffs.
<code>get_target_score(model)</code>	Compute the optimization target score (<code>self.target_metric</code>) with the cutoff being <code>self.cutoff</code> .

Attributes

get_score(*model*)

Compute the score with the cutoff being `self.cutoff`.

Parameters

model ([BaseRecommender](#)) – The evaluated recommender.

Returns

metric values.

Return type

`Dict[str, float]`

get_scores(*model*, *cutoffs*)

Compute the score with the specified cutoffs.

Parameters

- **model** ([BaseRecommender](#)) – The evaluated recommender.
- **cutoffs** (`List[int]`) – for each value in cutoff, the class computes the metric values.

Returns

The Resulting metric values. This time, the result will look like `{"ndcg@20": 0.35, "map@20": 0.2, ...}`.

Return type

`Dict[str, float]`

get_target_score(*model*)

Compute the optimization target score (`self.target_metric`) with the cutoff being `self.cutoff`.

Parameters

model ([BaseRecommender](#)) – The evaluated model.

Returns

The metric value.

Return type

`float`

1.4.2 Recommenders

<i>BaseRecommender</i>	The base class for all (hot) recommenders.
<i>TopPopRecommender</i>	A simple recommender system based on the popularity of the items in the training set (without any personalization).
<i>IALSRecommender</i>	Implementation of implicit Alternating Least Squares (iALS) or Weighted Matrix Factorization (WMF).
<i>P3alphaRecommender</i>	Recommendation with 3-steps random walk, proposed in
<i>RP3betaRecommender</i>	3-Path random walk with the item-popularity penalization:
<i>TruncatedSVDRecommender</i>	Use (randomized) SVD to factorize the input matrix into low-rank matrices.
<i>CosineKNNRecommender</i>	K-nearest neighbor recommender system based on cosine similarity.
<i>AsymmetricCosineKNNRecommender</i>	K-nearest neighbor recommender system based on asymmetric cosine similarity.
<i>JaccardKNNRecommender</i>	K-nearest neighbor recommender system based on Jaccard similarity.
<i>TverskyIndexKNNRecommender</i>	K-nearest neighbor recommender system based on Tversky Index.
<i>CosineUserKNNRecommender</i>	K-nearest neighbor recommender system based on cosine similarity.
<i>AsymmetricCosineUserKNNRecommender</i>	K-nearest neighbor recommender system based on asymmetric cosine similarity.
<i>SLIMRecommender</i>	SLIM with ElasticNet-type loss function:
<i>DenseSLIMRecommender</i>	Implementation of DenseSLIM or Embarrassingly Shallow AutoEncoder (EASE ^R).
<i>get_recommender_class</i>	Get recommender class from its class name.

irspack.recommenders.BaseRecommender

class irspack.recommenders.**BaseRecommender**(*X_train_all*, ****kwargs**)

Bases: object

The base class for all (hot) recommenders.

Parameters

- **X_train_all** (*csr_matrix/csc_matrix/np.ndarray*) – user/item interaction matrix. each row corresponds to a user’s interaction with items.
- **kwargs** (*Any*) –

__init__(*X_train_all*, ****kwargs**)

Parameters

- **X_train_all** (*Union[csr_matrix, csc_matrix]*) –
- **kwargs** (*Any*) –

Return type

None

Methods

<code>__init__(X_train_all, **kwargs)</code>	
<code>default_suggest_parameter(trial, fixed_params)</code>	
<code>from_config(X_train_all, config)</code>	
<code>get_score(user_indices)</code>	Compute the item recommendation score for a subset of users.
<code>get_score_block(begin, end)</code>	Compute the score for a block of the users.
<code>get_score_cold_user(X)</code>	Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix.
<code>get_score_cold_user_remove_seen(X)</code>	Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix.
<code>get_score_remove_seen(user_indices)</code>	Compute the item score and mask the item in the training set.
<code>get_score_remove_seen_block(begin, end)</code>	Compute the score for a block of the users, and mask the items in the training set.
<code>learn()</code>	Learns and returns itself.
<code>learn_with_optimizer(evaluator, trial[, ...])</code>	Learning procedures with early stopping and pruning.
<code>tune(data, evaluator[, n_trials, timeout, ...])</code>	Perform the optimization step.
<code>tune_with_study(study, data, evaluator[, ...])</code>	

Attributes

<code>config_class</code>	
<code>default_tune_range</code>	
<code>X_train_all</code>	The matrix to feed into recommender.

X_train_all: csr_matrix

The matrix to feed into recommender.

abstract get_score(*user_indices*)

Compute the item recommendation score for a subset of users.

Parameters

user_indices (*ndarray*) – The index defines the subset of users.

Returns

The item scores. Its shape will be (len(*user_indices*), self.n_items)

Return type

ndarray

get_score_block(*begin*, *end*)

Compute the score for a block of the users.

Parameters

- **begin** (*int*) – where the evaluated user block begins.
- **end** (*int*) – where the evaluated user block ends.

Returns

The item scores. Its shape will be (end - begin, self.n_items)

Return type

ndarray

get_score_cold_user(X)

Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix.

Parameters

X (*Union[csr_matrix, csc_matrix]*) – The profile user-item relation matrix for unseen users. Its number of rows is arbitrary, but the number of columns must be self.n_items.

Returns

Computed item scores for users. Its shape is equal to X.

Return type

ndarray

get_score_cold_user_remove_seen(X)

Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix. The score will then be masked by the input.

Parameters

X (*Union[csr_matrix, csc_matrix]*) – The profile user-item relation matrix for unseen users. Its number of rows is arbitrary, but the number of columns must be self.n_items.

Returns

Computed & masked item scores for users. Its shape is equal to X.

Return type

ndarray

get_score_remove_seen(user_indices)

Compute the item score and mask the item in the training set. Masked items will have the score -inf.

Parameters

user_indices (*ndarray*) – Specifies the subset of users.

Returns

The masked item scores. Its shape will be (len(user_indices), self.n_items)

Return type

ndarray

get_score_remove_seen_block(begin, end)

Compute the score for a block of the users, and mask the items in the training set. Masked items will have the score -inf.

Parameters

- **begin** (*int*) – where the evaluated user block begins.
- **end** (*int*) – where the evaluated user block ends.

Returns

The masked item scores. Its shape will be (end - begin, self.n_items)

Return type*ndarray***learn()**

Learns and returns itself.

Returns

The model after fitting process.

Parameters

self (*R*) –

Return type*R***learn_with_optimizer**(*evaluator, trial, max_epoch=128, validate_epoch=5, score_degradation_max=5*)

Learning procedures with early stopping and pruning.

Parameters

- **evaluator** (*Optional[[evaluation.Evaluator](#)]*) – The evaluator to measure the score.
- **trial** (*Optional[[Trial](#)]*) – The current optuna trial under the study (if any.)
- **max_epoch** (*int*) – Maximal number of epochs. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 128.
- **validate_epoch** (*int*) – The frequency of validation score measurement. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.
- **validate_epoch** – The frequency of validation score measurement. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.
- **score_degradation_max** (*int*) – Maximal number of allowed score degradation. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.

Return type*None*

classmethod tune(*data, evaluator, n_trials=20, timeout=None, data_suggest_function=None, parameter_suggest_function=None, fixed_params={}, random_seed=None, pruning_n_startup_trials=10, max_epoch=128, validate_epoch=5, score_degradation_max=5, logger=None*)

Perform the optimization step. *optuna.Study* object is created inside this function.

Parameters

- **data** (*Optional[Union[[csr_matrix](#), [csc_matrix](#)]]*) – The training data. You can also provide tunable parameter dependent training data by providing *data_suggest_function*. In that case, data must be *None*.
- **evaluator** ([evaluation.Evaluator](#)) – The validation evaluator that measures the performance of the recommenders.
- **n_trials** (*int*) – The number of expected trials (including pruned ones). Defaults to 20.
- **timeout** (*Optional[int]*) – If set to some value (in seconds), the study will exit after that time period. Note that the running trials is not interrupted, though. Defaults to *None*.
- **data_suggest_function** (*Optional[Callable[[[Trial](#)], Union[[csr_matrix](#), [csc_matrix](#)]]]*) – If not *None*, this must be a function which takes *optuna.Trial* as its argument and returns training data. Defaults to *None*.

- **parameter_suggest_function** (*Optional[Callable[[Trial], Dict[str, Any]]]*) – If not *None*, this must be a function which takes *optuna.Trial* as its argument and returns *Dict[str, Any]* (i.e., some keyword arguments of the recommender class). If *None*, *cls.default_suggest_parameter* will be used for the parameter suggestion. Defaults to *None*.
- **fixed_params** (*Dict[str, Any]*) – Fixed parameters passed to recommenders during the optimization procedure. This will replace the suggested parameter (either by *cls.default_suggest_parameter* or *parameter_suggest_function*). Defaults to *dict()*.
- **random_seed** (*Optional[int]*) – The random seed to control *optuna.samplers.TPESampler*. Defaults to *None*.
- **prunning_n_startup_trials** (*int*) – *n_startup_trials* argument passed to the constructor of *optuna.pruners.MedianPruner*.
- **max_epoch** (*int*) – The maximal number of epochs for the training. If iterative learning procedure is not available, this parameter will be ignored.
- **validate_epoch** (*int, optional*) – The frequency of validation score measurement. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.
- **score_degradation_max** (*int, optional*) – Maximal number of allowed score degradation. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5. Defaults to 5.
- **logger** (*Optional[Logger]*) –

Returns

A tuple that consists of

1. A dict containing the best paramaters. This dict can be passed to the recommender as ****kwargs**.
2. A *pandas.DataFrame* that contains the history of optimization.

Return type

Tuple[Dict[str, Any], DataFrame]

irspack.recommenders.TopPopRecommender

```
class irspack.recommenders.TopPopRecommender(X_train)
```

Bases: *BaseRecommender*

A simple recommender system based on the popularity of the items in the training set (without any personalization).

Parameters

- **Union[*scipy.sparse.csr_matrix* (*X_train*)** – Input interaction matrix.
- ***scipy.sparse.csc_matrix*]** – Input interaction matrix.
- **X_train** (*Union[csr_matrix, csc_matrix]*) –

__init__ (*X_train*)

Parameters

X_train (*Union[csr_matrix, csc_matrix]*) –

Methods

<code>__init__(X_train)</code>	
<code>default_suggest_parameter(trial, fixed_params)</code>	
<code>from_config(X_train_all, config)</code>	
<code>get_score(user_indices)</code>	Compute the item recommendation score for a subset of users.
<code>get_score_block(begin, end)</code>	Compute the score for a block of the users.
<code>get_score_cold_user(X)</code>	Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix.
<code>get_score_cold_user_remove_seen(X)</code>	Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix.
<code>get_score_remove_seen(user_indices)</code>	Compute the item score and mask the item in the training set.
<code>get_score_remove_seen_block(begin, end)</code>	Compute the score for a block of the users, and mask the items in the training set.
<code>learn()</code>	Learns and returns itself.
<code>learn_with_optimizer(evaluator, trial[, ...])</code>	Learning procedures with early stopping and pruning.
<code>tune(data, evaluator[, n_trials, timeout, ...])</code>	Perform the optimization step.
<code>tune_with_study(study, data, evaluator[, ...])</code>	

Attributes

<code>default_tune_range</code>
<code>score</code>
<code>score_</code>

X_train_all: `sps.csr_matrix`

The matrix to feed into recommender.

get_score(*user_indices*)

Compute the item recommendation score for a subset of users.

Parameters

user_indices (*ndarray*) – The index defines the subset of users.

Returns

The item scores. Its shape will be (len(*user_indices*), self.n_items)

Return type

ndarray

get_score_block(*begin, end*)

Compute the score for a block of the users.

Parameters

- **begin** (*int*) – where the evaluated user block begins.
- **end** (*int*) – where the evaluated user block ends.

Returns

The item scores. Its shape will be (end - begin, self.n_items)

Return type

ndarray

get_score_cold_user(X)

Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix.

Parameters

X (*Union[csr_matrix, csc_matrix]*) – The profile user-item relation matrix for unseen users. Its number of rows is arbitrary, but the number of columns must be self.n_items.

Returns

Computed item scores for users. Its shape is equal to X.

Return type

ndarray

get_score_cold_user_remove_seen(X)

Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix. The score will then be masked by the input.

Parameters

X (*Union[csr_matrix, csc_matrix]*) – The profile user-item relation matrix for unseen users. Its number of rows is arbitrary, but the number of columns must be self.n_items.

Returns

Computed & masked item scores for users. Its shape is equal to X.

Return type

ndarray

get_score_remove_seen(user_indices)

Compute the item score and mask the item in the training set. Masked items will have the score -inf.

Parameters

user_indices (*ndarray*) – Specifies the subset of users.

Returns

The masked item scores. Its shape will be (len(user_indices), self.n_items)

Return type

ndarray

get_score_remove_seen_block(begin, end)

Compute the score for a block of the users, and mask the items in the training set. Masked items will have the score -inf.

Parameters

- **begin** (*int*) – where the evaluated user block begins.
- **end** (*int*) – where the evaluated user block ends.

Returns

The masked item scores. Its shape will be (end - begin, self.n_items)

Return type*ndarray***learn()**

Learns and returns itself.

Returns

The model after fitting process.

Parameters

self (*R*) –

Return type*R***learn_with_optimizer**(*evaluator, trial, max_epoch=128, validate_epoch=5, score_degradation_max=5*)

Learning procedures with early stopping and pruning.

Parameters

- **evaluator** (*Optional[[evaluation.Evaluator](#)]*) – The evaluator to measure the score.
- **trial** (*Optional[[Trial](#)]*) – The current optuna trial under the study (if any.)
- **max_epoch** (*int*) – Maximal number of epochs. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 128.
- **validate_epoch** (*int*) – The frequency of validation score measurement. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.
- **validate_epoch** – The frequency of validation score measurement. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.
- **score_degradation_max** (*int*) – Maximal number of allowed score degradation. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.

Return type*None*

classmethod tune(*data, evaluator, n_trials=20, timeout=None, data_suggest_function=None, parameter_suggest_function=None, fixed_params={}, random_seed=None, pruning_n_startup_trials=10, max_epoch=128, validate_epoch=5, score_degradation_max=5, logger=None*)

Perform the optimization step. *optuna.Study* object is created inside this function.

Parameters

- **data** (*Optional[Union[[csr_matrix](#), [csc_matrix](#)]]*) – The training data. You can also provide tunable parameter dependent training data by providing *data_suggest_function*. In that case, data must be *None*.
- **evaluator** ([evaluation.Evaluator](#)) – The validation evaluator that measures the performance of the recommenders.
- **n_trials** (*int*) – The number of expected trials (including pruned ones). Defaults to 20.
- **timeout** (*Optional[int]*) – If set to some value (in seconds), the study will exit after that time period. Note that the running trials is not interrupted, though. Defaults to *None*.
- **data_suggest_function** (*Optional[Callable[[[Trial](#)], Union[[csr_matrix](#), [csc_matrix](#)]]]*) – If not *None*, this must be a function which takes *optuna.Trial* as its argument and returns training data. Defaults to *None*.

- **parameter_suggest_function** (*Optional[Callable[[Trial], Dict[str, Any]]]*) – If not *None*, this must be a function which takes *optuna.Trial* as its argument and returns *Dict[str, Any]* (i.e., some keyword arguments of the recommender class). If *None*, *cls.default_suggest_parameter* will be used for the parameter suggestion. Defaults to *None*.
- **fixed_params** (*Dict[str, Any]*) – Fixed parameters passed to recommenders during the optimization procedure. This will replace the suggested parameter (either by *cls.default_suggest_parameter* or *parameter_suggest_function*). Defaults to *dict()*.
- **random_seed** (*Optional[int]*) – The random seed to control *optuna.samplers.TPESampler*. Defaults to *None*.
- **pruning_n_startup_trials** (*int*) – *n_startup_trials* argument passed to the constructor of *optuna.pruners.MedianPruner*.
- **max_epoch** (*int*) – The maximal number of epochs for the training. If iterative learning procedure is not available, this parameter will be ignored.
- **validate_epoch** (*int, optional*) – The frequency of validation score measurement. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.
- **score_degradation_max** (*int, optional*) – Maximal number of allowed score degradation. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5. Defaults to 5.
- **logger** (*Optional[Logger]*) –

Returns

A tuple that consists of

1. A dict containing the best paramaters. This dict can be passed to the recommender as ****kwargs**.
2. A *pandas.DataFrame* that contains the history of optimization.

Return type

Tuple[Dict[str, Any], DataFrame]

irspack.recommenders.IALSRecommender

```
class irspack.recommenders.IALSRecommender(X_train_all, n_components=20, alpha0=0.0, reg=0.001,
                                             nu=1.0, confidence_scaling='none', epsilon=1.0,
                                             init_std=0.1, solver_type='CG', max_cg_steps=3,
                                             ialspp_subspace_dimension=64, loss_type='IALSPP',
                                             nu_star=None, random_seed=42, n_threads=None,
                                             train_epochs=16, prediction_time_max_cg_steps=5,
                                             prediction_time_ialspp_iteration=7)
```

Bases: *BaseRecommenderWithEarlyStopping*, *BaseRecommenderWithUserEmbedding*, *BaseRecommenderWithItemEmbedding*

Implementation of implicit Alternating Least Squares (iALS) or Weighted Matrix Factorization (WMF).

By default, it tries to minimize the following loss:

$$\frac{1}{2} \sum_{u,i \in S} c_{ui} (\mathbf{u}_u \cdot \mathbf{v}_i - 1)^2 + \frac{\alpha_0}{2} \sum_{u,i} (\mathbf{u}_u \cdot \mathbf{v}_i)^2 + \frac{\text{reg}}{2} \left(\sum_u (\alpha_0 I + N_u)^\nu \|\mathbf{u}_u\|^2 + \sum_i (\alpha_0 U + N_i)^\nu \|\mathbf{v}_i\|^2 \right)$$

where S denotes the set of all pairs where X_{ui} is non-zero.

See the seminal paper:

- [Collaborative filtering for implicit feedback datasets](#)

By default it uses a conjugate gradient descent version:

- [Applications of the conjugate gradient method for implicit feedback collaborative filtering](#)

The loss above is slightly different from the original version. See the following paper for the loss used here

- [Revisiting the Performance of iALS on Item Recommendation Benchmarks](#)

Parameters

- **X_train_all** (*Union[scipy.sparse.csr_matrix, scipy.sparse.csc_matrix]*) – Input interaction matrix.
- **n_components** (*int, optional*) – The dimension for latent factor. Defaults to 20.
- **alpha0** (*float, optional*) – The “unobserved” weight.
- **reg** (*float, optional*) – Regularization coefficient for both user & item factors. Defaults to 1e-3.
- **nu** (*float, optional*) – Controls frequency regularization introduced in the paper, “Revisiting the Performance of iALS on Item Recommendation Benchmarks”.
- **confidence_scaling** (*str, optional*) – Specifies how to scale confidence scaling c_{ui} . Must be either “none” or “log”. If “none”, the non-zero (not-necessarily 1) X_{ui} yields

$$c_{ui} = A + X_{ui}$$

If “log”,

$$c_{ui} = A + \log(1 + X_{ui}/\epsilon)$$

The constant A above will be 0 if `loss_type` is “IALSPP”, α_0 if `loss_type` is “ORIGINAL”.

Defaults to “none”.

- **epsilon** (*float, optional*) – The ϵ parameter for log-scaling described above. Will not have any effect if `confidence_scaling` is “none”. Defaults to 1.0f.
- **init_std** (*float, optional*) – Standard deviation for initialization normal distribution. The actual std for each user/item vector components are scaled by $1 / n_components ** .5$. Defaults to 0.1.
- **solver_type** (“CHOLSKY” | “CG” | “IALSPP”, *optional*) – Which solver to use. Defaults to “CG”.
- **max_cg_steps** (*int, optional*) – Maximal number of conjugate gradient descent steps during the training time. Defaults to 3. Used only when `solver_type`==“CG”. By increasing this parameter, the result will be closer to Cholesky decomposition method (i.e., when `solver_type` == “CHOLSKY”), but it will take longer time.
- **ialspp_subspace_dimension** (*int, optional*) – The subspace dimension of iALS++ (ignored if the `solver_type` is not “IALSPP”). If this value is 1, specialized implementation described in [Fast Matrix Factorization for Online Recommendation with Implicit Feedback](#) will be used instead. Defaults to 64.

- **loss_type** (*Literal*["IALSPP", "ORIGINAL"], *optional*) – Specifies the subtle difference between iALS++ vs Original Loss.
- **nu_star** (*Optional*[*float*], *optional*) – If not *None*, used as the reference scale for nu described in the “Revisiting...” paper. Defaults to *None*.
- **random_seed** (*int*, *optional*) – The random seed to initialize the parameters.
- **n_threads** (*Optional*[*int*], *optional*) – Specifies the number of threads to use for the computation. If *None*, the environment variable "IRSPACK_NUM_THREADS_DEFAULT" will be looked up, and if the variable is not set, it will be set to `os.cpu_count()`. Defaults to *None*.
- **train_epochs** (*int*, *optional*) – Maximal number of epochs. Defaults to 16.
- **prediction_time_max_cg_steps** (*int*, *optional*) – Maximal number of conjugate gradient descent steps during the prediction time, i.e., the case when a user unseen at the training time is given as a history matrix. Defaults to 5.
- **prediction_time_ialspp_iteration** (*int*) –

Examples

```
>>> from irspack import IALSRecommender, rowwise_train_test_split, Evaluator
>>> from irspack.utils.sample_data import mf_example_data
>>> X = mf_example_data(100, 30, random_state=1)
>>> X_train, X_test = rowwise_train_test_split(X, random_state=0)
>>> rec = IALSRecommender(X_train)
>>> rec.learn()
>>> evaluator=Evaluator(X_test)
>>> print(evaluator.get_scores(rec, [20]))
OrderedDict([('hit@20', 1.0), ('recall@20', 0.9003412698412698), ('ndcg@20', 0.
→ 6175493479217139), ('map@20', 0.3848785870622406), ('precision@20', 0.3385), (
→ 'gini_index@20', 0.0814), ('entropy@20', 3.382497875272383), ('appeared_item@20',
→ 30.0)])
```

```
__init__(X_train_all, n_components=20, alpha0=0.0, reg=0.001, nu=1.0, confidence_scaling='none',
epsilon=1.0, init_std=0.1, solver_type='CG', max_cg_steps=3, ialspp_subspace_dimension=64,
loss_type='IALSPP', nu_star=None, random_seed=42, n_threads=None, train_epochs=16,
prediction_time_max_cg_steps=5, prediction_time_ialspp_iteration=7)
```

Parameters

- **X_train_all** (*Union*[*csr_matrix*, *csc_matrix*]) –
- **n_components** (*int*) –
- **alpha0** (*float*) –
- **reg** (*float*) –
- **nu** (*float*) –
- **confidence_scaling** (*str*) –
- **epsilon** (*float*) –
- **init_std** (*float*) –
- **solver_type** (*typing_extensions.Literal*[*CG*, *CHOLSKY*, *IALSPP*]) –

- `max_cg_steps(int)` –
- `ialspp_subspace_dimension(int)` –
- `loss_type(typing_extensions.Literal[IALSPP, ORIGINAL])` –
- `nu_star(Optional[float])` –
- `random_seed(int)` –
- `n_threads(Optional[int])` –
- `train_epochs(int)` –
- `prediction_time_max_cg_steps(int)` –
- `prediction_time_ialspp_iteration(int)` –

Return type

None

Methods

<code>__init__(X_train_all[, n_components, ...])</code>	
<code>compute_item_embedding(X)</code>	Given an unknown items' interaction with known user, computes the latent factors of the items by least square (fixing user embeddings).
<code>compute_user_embedding(X)</code>	Given an unknown users' interaction with known items, computes the latent factors of the users by least square (fixing item embeddings).
<code>default_suggest_parameter(trial, fixed_params)</code>	
<code>from_config(X_train_all, config)</code>	
<code>get_item_embedding()</code>	Get item embedding vectors.
<code>get_score(user_indices)</code>	Compute the item recommendation score for a subset of users.
<code>get_score_block(begin, end)</code>	Compute the score for a block of the users.
<code>get_score_cold_user(X)</code>	Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix.
<code>get_score_cold_user_remove_seen(X)</code>	Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix.
<code>get_score_from_item_embedding(user_indices, ...)</code>	
<code>get_score_from_user_embedding(user_embedding)</code>	Compute the item score from user embedding.
<code>get_score_remove_seen(user_indices)</code>	Compute the item score and mask the item in the training set.
<code>get_score_remove_seen_block(begin, end)</code>	Compute the score for a block of the users, and mask the items in the training set.
<code>get_user_embedding()</code>	Get user embedding vectors.
<code>learn()</code>	Learns and returns itself.
<code>learn_with_optimizer(evaluator, trial[, ...])</code>	Learning procedures with early stopping and pruning.
<code>load_state()</code>	
<code>run_epoch()</code>	
<code>save_state()</code>	
<code>start_learning()</code>	
<code>tune(data, evaluator[, n_trials, timeout, ...])</code>	Perform the optimization step.
<code>tune_doubling_dimension(data, evaluator, ...)</code>	Perform tuning gradually doubling <i>n_components</i> .
<code>tune_with_study(study, data, evaluator[, ...])</code>	

Attributes

`default_tune_range`

`trainer_as_ials`

X_train_all: `sps.csr_matrix`

The matrix to feed into recommender.

compute_item_embedding(*X*)

Given an unknown items' interaction with known user, computes the latent factors of the items by least square (fixing user embeddings).

Parameters

X (*Union[csr_matrix, csc_matrix]*) – The interaction history of the new users. **X**. `shape[0]` must be equal to `self.n_users`.

Return type

ndarray

compute_user_embedding(*X*)

Given an unknown users' interaction with known items, computes the latent factors of the users by least square (fixing item embeddings).

Parameters

X (*Union[csr_matrix, csc_matrix]*) – The interaction history of the new users. **X**. `shape[1]` must be equal to `self.n_items`.

Return type

ndarray

get_item_embedding()

Get item embedding vectors.

Returns

The latent vector representation of items. Its number of rows is equal to the number of the items.

Return type

ndarray

get_score(*user_indices*)

Compute the item recommendation score for a subset of users.

Parameters

user_indices (*ndarray*) – The index defines the subset of users.

Returns

The item scores. Its shape will be `(len(user_indices), self.n_items)`

Return type

ndarray

get_score_block(*begin, end*)

Compute the score for a block of the users.

Parameters

- **begin** (*int*) – where the evaluated user block begins.

- **end** (*int*) – where the evaluated user block ends.

Returns

The item scores. Its shape will be (end - begin, self.n_items)

Return type

ndarray

get_score_cold_user(*X*)

Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix.

Parameters

X (*Union[csr_matrix, csc_matrix]*) – The profile user-item relation matrix for unseen users. Its number of rows is arbitrary, but the number of columns must be self.n_items.

Returns

Computed item scores for users. Its shape is equal to X.

Return type

ndarray

get_score_cold_user_remove_seen(*X*)

Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix. The score will then be masked by the input.

Parameters

X (*Union[csr_matrix, csc_matrix]*) – The profile user-item relation matrix for unseen users. Its number of rows is arbitrary, but the number of columns must be self.n_items.

Returns

Computed & masked item scores for users. Its shape is equal to X.

Return type

ndarray

get_score_from_user_embedding(*user_embedding*)

Compute the item score from user embedding. Mainly used for cold-start scenario.

Parameters

user_embedding (*ndarray*) – Latent user representation obtained elsewhere.

Returns

The score array. Its shape will be (user_embedding.shape[0], self.n_items)

Return type

DenseScoreArray

get_score_remove_seen(*user_indices*)

Compute the item score and mask the item in the training set. Masked items will have the score -inf.

Parameters

user_indices (*ndarray*) – Specifies the subset of users.

Returns

The masked item scores. Its shape will be (len(user_indices), self.n_items)

Return type

ndarray

get_score_remove_seen_block(begin, end)

Compute the score for a block of the users, and mask the items in the training set. Masked items will have the score -inf.

Parameters

- **begin** (*int*) – where the evaluated user block begins.
- **end** (*int*) – where the evaluated user block ends.

Returns

The masked item scores. Its shape will be (end - begin, self.n_items)

Return type

ndarray

get_user_embedding()

Get user embedding vectors.

Returns

The latent vector representation of users. Its number of rows is equal to the number of the users.

Return type

ndarray

learn()

Learns and returns itself.

Returns

The model after fitting process.

Parameters

self (*R*) –

Return type

R

learn_with_optimizer(evaluator, trial, max_epoch=128, validate_epoch=5, score_degradation_max=5)

Learning procedures with early stopping and pruning.

Parameters

- **evaluator** (*Optional[[evaluation.Evaluator](#)]*) – The evaluator to measure the score.
- **trial** (*Optional[Trial]*) – The current optuna trial under the study (if any.)
- **max_epoch** (*int*) – Maximal number of epochs. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 128.
- **validate_epoch** (*int*) – The frequency of validation score measurement. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.
- **validate_epoch** – The frequency of validation score measurement. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.
- **score_degradation_max** (*int*) – Maximal number of allowed score degradation. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.

Return type

None

```
classmethod tune(data, evaluator, n_trials=20, timeout=None, data_suggest_function=None,
                 parameter_suggest_function=None, fixed_params={}, random_seed=None,
                 prunning_n_startup_trials=10, max_epoch=16, validate_epoch=1,
                 score_degradation_max=3, logger=None)
```

Perform the optimization step. *optuna.Study* object is created inside this function.

Parameters

- **data** (*Optional[Union[csr_matrix, csc_matrix]]*) – The training data. You can also provide tunable parameter dependent training data by providing *data_suggest_function*. In that case, data must be *None*.
- **evaluator** (*evaluation.Evaluator*) – The validation evaluator that measures the performance of the recommenders.
- **n_trials** (*int*) – The number of expected trials (including pruned ones). Defaults to 20.
- **timeout** (*Optional[int]*) – If set to some value (in seconds), the study will exit after that time period. Note that the running trials is not interrupted, though. Defaults to *None*.
- **data_suggest_function** (*Optional[Callable[[Trial], Union[csr_matrix, csc_matrix]]]*) – If not *None*, this must be a function which takes *optuna.Trial* as its argument and returns training data. Defaults to *None*.
- **parameter_suggest_function** (*Optional[Callable[[Trial], Dict[str, Any]]]*) – If not *None*, this must be a function which takes *optuna.Trial* as its argument and returns *Dict[str, Any]* (i.e., some keyword arguments of the recommender class). If *None*, *cls.default_suggest_parameter* will be used for the parameter suggestion. Defaults to *None*.
- **fixed_params** (*Dict[str, Any]*) – Fixed parameters passed to recommenders during the optimization procedure. This will replace the suggested parameter (either by *cls.default_suggest_parameter* or *parameter_suggest_function*). Defaults to *dict()*.
- **random_seed** (*Optional[int]*) – The random seed to control *optuna.samplers.TPESampler*. Defaults to *None*.
- **prunning_n_startup_trials** (*int*) – *n_startup_trials* argument passed to the constructor of *optuna.pruners.MedianPruner*.
- **max_epoch** (*int*) – The maximal number of epochs for the training. If iterative learning procedure is not available, this parameter will be ignored.
- **validate_epoch** (*int, optional*) – The frequency of validation score measurement. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.
- **score_degradation_max** (*int, optional*) – Maximal number of allowed score degradation. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5. Defaults to 5.
- **logger** (*Optional[Logger]*) –

Returns

A tuple that consists of

1. A dict containing the best paramaters. This dict can be passed to the recommender as ****kwargs**.
2. A *pandas.DataFrame* that contains the history of optimization.

Return type*Tuple[Dict[str, Any], DataFrame]*

classmethod **tune_doubling_dimension**(*data, evaluator, initial_dimension, maximal_dimension, storage=None, study_name_prefix=None, n_trials_initial=40, n_trials_following=20, n_startup_trials_initial=10, n_startup_trials_following=5, max_epoch=16, validate_epoch=1, score_degradation_max=3, neighborhood_scale=3.0, suggest_function_initial=None, random_seed=None*)

Perform tuning gradually doubling *n_components*. Typically, with the initial *n_components*, the search will be more exhaustive, and with larger *n_components*, less exploration will be done around previously found parameters. This strategy is described in [Revisiting the Performance of iALS on Item Recommendation Benchmarks](#).

Parameters

- **initial_dimension** (*int*) – The initial dimension.
- **maximal_dimension** (*int*) – The maximal (inclusive) dimension to be tried.
- **storage** (*Optional[RDBStorage]*) – The storage where multiple *optuna.Study* will be created corresponding to the various dimensions. If *None*, all *Study* will be created in-memory.
- **study_name_prefix** (*Optional[str]*) – The prefix for the names of *optuna.Study*. For dimension *d*, the full name of the *Study* will be “{study_name_prefix}_{d}”. If *None*, we will use a random string for this prefix.
- **n_trials_initial** (*int*) – The number of trials for the initial dimension.
- **n_trials_following** (*int*) – The number of trials for the following dimensions.
- **n_startup_trials_initial** (*int*) – Passed on to *n_startup_trials* argument of *optuna.pruners.MedianPruner* in the initial *optuna.Study*. Defaults to 10.
- **n_startup_trials_following** (*int*) – Passed on to *n_startup_trials* argument of *optuna.pruners.MedianPruner* in the following *optuna.Study*. Defaults to 5.
- **neighborhood_scale** (*float*) – *alpha_0* and *reg* parameters will be searched within the log-uniform range [*previous_dimension_result* / *neighborhood_scale*, *previous_dimension_result* * *neighborhood_scale*]. Defaults to 3.0
- **suggest_overwrite_initial** – Overwrites the suggestion parameters in the initial *optuna.Study*. Defaults to [].
- **random_seed** (*Optional[int]*) – The random seed to control *optuna.samplers.TPESampler*. Defaults to *None*.
- **data** (*Union[csr_matrix, csc_matrix]*) –
- **evaluator** (*Evaluator*) –
- **max_epoch** (*int*) –
- **validate_epoch** (*int*) –
- **score_degradation_max** (*int*) –
- **suggest_function_initial** (*Optional[Callable[[Trial], Dict[str, Any]]]*) –

Returns

A tuple that consists of

1. A dict containing the best paramaters. This dict can be passed to the recommender as `**kwargs`.
2. A `pandas.DataFrame` that contains the history of optimization for all dimensions.

Return type

`Tuple[Dict[str, Any], DataFrame]`

`irspack.recommenders.P3alphaRecommender`

```
class irspack.recommenders.P3alphaRecommender(X_train_all, alpha=1, top_k=None,
                                              normalize_weight=False, n_threads=None)
```

Bases: `BaseSimilarityRecommender`

Recommendation with 3-steps random walk, proposed in

- [Random Walks in Recommender Systems: Exact Computation and Simulations](#)

The version here implements its view as KNN-based method, as pointed out in

- [A Troubling Analysis of Reproducibility and Progress in Recommender Systems Research](#)

Parameters

- **X_train_all** (`Union[scipy.sparse.csr_matrix, scipy.sparse.csc_matrix]`) – Input interaction matrix.
- **alpha** (`float, optional`) – The power to which `X_train_all` is exponentiated. Defaults to 1. Note that this has no effect if all the entries in `X_train_all` are equal.
- **top_k** (`Optional[int], optional`) – Maximal number of non-zero entries retained for each column of the similarity matrix `W`.
- **normalize_weight** (`bool, optional`) – Whether to perform row-wise normalization of `W`. Defaults to `False`.
- **n_threads** (`Optional[int], optional`) – Specifies the number of threads to use for the computation. If `None`, the environment variable "IRSPACK_NUM_THREADS_DEFAULT" will be looked up, and if the variable is not set, it will be set to `os.cpu_count()`. Defaults to `None`.

```
__init__(X_train_all, alpha=1, top_k=None, normalize_weight=False, n_threads=None)
```

Parameters

- **X_train_all** (`Union[csr_matrix, csc_matrix]`) –
- **alpha** (`float`) –
- **top_k** (`Optional[int]`) –
- **normalize_weight** (`bool`) –
- **n_threads** (`Optional[int]`) –

Methods

<code>__init__(X_train_all[, alpha, top_k, ...])</code>	
<code>default_suggest_parameter(trial, fixed_params)</code>	
<code>from_config(X_train_all, config)</code>	
<code>get_score(user_indices)</code>	Compute the item recommendation score for a subset of users.
<code>get_score_block(begin, end)</code>	Compute the score for a block of the users.
<code>get_score_cold_user(X)</code>	Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix.
<code>get_score_cold_user_remove_seen(X)</code>	Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix.
<code>get_score_remove_seen(user_indices)</code>	Compute the item score and mask the item in the training set.
<code>get_score_remove_seen_block(begin, end)</code>	Compute the score for a block of the users, and mask the items in the training set.
<code>learn()</code>	Learns and returns itself.
<code>learn_with_optimizer(evaluator, trial[, ...])</code>	Learning procedures with early stopping and pruning.
<code>tune(data, evaluator[, n_trials, timeout, ...])</code>	Perform the optimization step.
<code>tune_with_study(study, data, evaluator[, ...])</code>	

Attributes

<code>\bar{W}</code>	The computed item-item similarity weight matrix.
<code>default_tune_range</code>	

property \bar{W} : Union[csr_matrix, csc_matrix, ndarray]

The computed item-item similarity weight matrix.

$X_{\text{train_all}}$: sps.csr_matrix

The matrix to feed into recommender.

`get_score(user_indices)`

Compute the item recommendation score for a subset of users.

Parameters

`user_indices` (ndarray) – The index defines the subset of users.

Returns

The item scores. Its shape will be (len(*user_indices*), self.n_items)

Return type

ndarray

`get_score_block(begin, end)`

Compute the score for a block of the users.

Parameters

- **begin** (*int*) – where the evaluated user block begins.
- **end** (*int*) – where the evaluated user block ends.

Returns

The item scores. Its shape will be (end - begin, self.n_items)

Return type

ndarray

get_score_cold_user(X)

Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix.

Parameters

X (*Union[csr_matrix, csc_matrix]*) – The profile user-item relation matrix for unseen users. Its number of rows is arbitrary, but the number of columns must be self.n_items.

Returns

Computed item scores for users. Its shape is equal to X.

Return type

ndarray

get_score_cold_user_remove_seen(X)

Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix. The score will then be masked by the input.

Parameters

X (*Union[csr_matrix, csc_matrix]*) – The profile user-item relation matrix for unseen users. Its number of rows is arbitrary, but the number of columns must be self.n_items.

Returns

Computed & masked item scores for users. Its shape is equal to X.

Return type

ndarray

get_score_remove_seen(user_indices)

Compute the item score and mask the item in the training set. Masked items will have the score -inf.

Parameters

user_indices (*ndarray*) – Specifies the subset of users.

Returns

The masked item scores. Its shape will be (len(user_indices), self.n_items)

Return type

ndarray

get_score_remove_seen_block(begin, end)

Compute the score for a block of the users, and mask the items in the training set. Masked items will have the score -inf.

Parameters

- **begin** (*int*) – where the evaluated user block begins.
- **end** (*int*) – where the evaluated user block ends.

Returns

The masked item scores. Its shape will be (end - begin, self.n_items)

Return type*ndarray***learn()**

Learns and returns itself.

Returns

The model after fitting process.

Parameters

self (*R*) –

Return type*R***learn_with_optimizer**(*evaluator, trial, max_epoch=128, validate_epoch=5, score_degradation_max=5*)

Learning procedures with early stopping and pruning.

Parameters

- **evaluator** (*Optional[[evaluation.Evaluator](#)]*) – The evaluator to measure the score.
- **trial** (*Optional[Trial]*) – The current optuna trial under the study (if any.)
- **max_epoch** (*int*) – Maximal number of epochs. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 128.
- **validate_epoch** (*int*) – The frequency of validation score measurement. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.
- **validate_epoch** – The frequency of validation score measurement. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.
- **score_degradation_max** (*int*) – Maximal number of allowed score degradation. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.

Return type*None*

classmethod tune(*data, evaluator, n_trials=20, timeout=None, data_suggest_function=None, parameter_suggest_function=None, fixed_params={}, random_seed=None, pruning_n_startup_trials=10, max_epoch=128, validate_epoch=5, score_degradation_max=5, logger=None*)

Perform the optimization step. *optuna.Study* object is created inside this function.

Parameters

- **data** (*Optional[Union[[csr_matrix](#), [csc_matrix](#)]]*) – The training data. You can also provide tunable parameter dependent training data by providing *data_suggest_function*. In that case, data must be *None*.
- **evaluator** ([evaluation.Evaluator](#)) – The validation evaluator that measures the performance of the recommenders.
- **n_trials** (*int*) – The number of expected trials (including pruned ones). Defaults to 20.
- **timeout** (*Optional[int]*) – If set to some value (in seconds), the study will exit after that time period. Note that the running trials is not interrupted, though. Defaults to *None*.
- **data_suggest_function** (*Optional[Callable[[[Trial](#)], Union[[csr_matrix](#), [csc_matrix](#)]]]*) – If not *None*, this must be a function which takes *optuna.Trial* as its argument and returns training data. Defaults to *None*.

- **parameter_suggest_function** (*Optional[Callable[[Trial], Dict[str, Any]]]*) – If not *None*, this must be a function which takes *optuna.Trial* as its argument and returns *Dict[str, Any]* (i.e., some keyword arguments of the recommender class). If *None*, *cls.default_suggest_parameter* will be used for the parameter suggestion. Defaults to *None*.
- **fixed_params** (*Dict[str, Any]*) – Fixed parameters passed to recommenders during the optimization procedure. This will replace the suggested parameter (either by *cls.default_suggest_parameter* or *parameter_suggest_function*). Defaults to *dict()*.
- **random_seed** (*Optional[int]*) – The random seed to control *optuna.samplers.TPESampler*. Defaults to *None*.
- **prunning_n_startup_trials** (*int*) – *n_startup_trials* argument passed to the constructor of *optuna.pruners.MedianPruner*.
- **max_epoch** (*int*) – The maximal number of epochs for the training. If iterative learning procedure is not available, this parameter will be ignored.
- **validate_epoch** (*int, optional*) – The frequency of validation score measurement. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.
- **score_degradation_max** (*int, optional*) – Maximal number of allowed score degradation. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5. Defaults to 5.
- **logger** (*Optional[Logger]*) –

Returns

A tuple that consists of

1. A dict containing the best paramaters. This dict can be passed to the recommender as ****kwargs**.
2. A *pandas.DataFrame* that contains the history of optimization.

Return type

Tuple[Dict[str, Any], DataFrame]

irspack.recommenders.RP3betaRecommender

```
class irspack.recommenders.RP3betaRecommender(X_train_all, alpha=1, beta=0.6, top_k=None,
                                              normalize_weight=False, n_threads=None)
```

Bases: *BaseSimilarityRecommender*

3-Path random walk with the item-popularity penalization:

- [Updatable, Accurate, Diverse, and Scalable Recommendations for Interactive Applications](#)

The version here implements its view as KNN-based method, as pointed out in

- [A Troubling Analysis of Reproducibility and Progress in Recommender Systems Research](#)

Parameters

- **X_train_all** (*Union[scipy.sparse.csr_matrix, scipy.sparse.csc_matrix]*) – Input interaction matrix.
- **alpha** (*float, optional*) – The power to which *X_train_all* is exponentiated. Defaults to 1. Note that this has no effect if all the entries in *X_train_all* are equal.

- **beta** (*float, optional*) – Specifies how the user->item transition probability will be penalized by the popularity. Defaults to 0.6.
- **top_k** (*Optional[int], optional*) – Maximal number of non-zero entries retained
- **W.** (*for each column of the similarity matrix*) –
- **normalize_weight** (*bool, optional*) – Whether to perform row-wise normalization of W. Defaults to False.
- **n_threads** (*Optional[int], optional*) – Specifies the number of threads to use for the computation. If None, the environment variable "IRSPACK_NUM_THREADS_DEFAULT" will be looked up, and if the variable is not set, it will be set to `os.cpu_count()`. Defaults to None.

`__init__(X_train_all, alpha=1, beta=0.6, top_k=None, normalize_weight=False, n_threads=None)`

Parameters

- **X_train_all** (*Union[csr_matrix, csc_matrix]*) –
- **alpha** (*float*) –
- **beta** (*float*) –
- **top_k** (*Optional[int]*) –
- **normalize_weight** (*bool*) –
- **n_threads** (*Optional[int]*) –

Methods

<code>__init__(X_train_all[, alpha, beta, top_k, ...])</code>	
<code>default_suggest_parameter(trial, fixed_params)</code>	
<code>from_config(X_train_all, config)</code>	
<code>get_score(user_indices)</code>	Compute the item recommendation score for a subset of users.
<code>get_score_block(begin, end)</code>	Compute the score for a block of the users.
<code>get_score_cold_user(X)</code>	Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix.
<code>get_score_cold_user_remove_seen(X)</code>	Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix.
<code>get_score_remove_seen(user_indices)</code>	Compute the item score and mask the item in the training set.
<code>get_score_remove_seen_block(begin, end)</code>	Compute the score for a block of the users, and mask the items in the training set.
<code>learn()</code>	Learns and returns itself.
<code>learn_with_optimizer(evaluator, trial[, ...])</code>	Learning procedures with early stopping and pruning.
<code>tune(data, evaluator[, n_trials, timeout, ...])</code>	Perform the optimization step.
<code>tune_with_study(study, data, evaluator[, ...])</code>	

Attributes

<i>W</i>	The computed item-item similarity weight matrix.
<code>default_tune_range</code>	

property W: Union[csr_matrix, csc_matrix, ndarray]

The computed item-item similarity weight matrix.

X_train_all: sps.csr_matrix

The matrix to feed into recommender.

get_score(*user_indices*)

Compute the item recommendation score for a subset of users.

Parameters

user_indices (*ndarray*) – The index defines the subset of users.

Returns

The item scores. Its shape will be (len(*user_indices*), self.n_items)

Return type

ndarray

get_score_block(*begin*, *end*)

Compute the score for a block of the users.

Parameters

- **begin** (*int*) – where the evaluated user block begins.
- **end** (*int*) – where the evaluated user block ends.

Returns

The item scores. Its shape will be (end - begin, self.n_items)

Return type

ndarray

get_score_cold_user(*X*)

Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix.

Parameters

X (*Union[csr_matrix, csc_matrix]*) – The profile user-item relation matrix for unseen users. Its number of rows is arbitrary, but the number of columns must be self.n_items.

Returns

Computed item scores for users. Its shape is equal to X.

Return type

ndarray

get_score_cold_user_remove_seen(*X*)

Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix. The score will then be masked by the input.

Parameters

X (*Union[csr_matrix, csc_matrix]*) – The profile user-item relation matrix for unseen users. Its number of rows is arbitrary, but the number of columns must be self.n_items.

Returns

Computed & masked item scores for users. Its shape is equal to X.

Return type

ndarray

get_score_remove_seen(*user_indices*)

Compute the item score and mask the item in the training set. Masked items will have the score -inf.

Parameters

user_indices (*ndarray*) – Specifies the subset of users.

Returns

The masked item scores. Its shape will be (len(*user_indices*), self.n_items)

Return type

ndarray

get_score_remove_seen_block(*begin*, *end*)

Compute the score for a block of the users, and mask the items in the training set. Masked items will have the score -inf.

Parameters

- **begin** (*int*) – where the evaluated user block begins.
- **end** (*int*) – where the evaluated user block ends.

Returns

The masked item scores. Its shape will be (end - begin, self.n_items)

Return type

ndarray

learn()

Learns and returns itself.

Returns

The model after fitting process.

Parameters

self (*R*) –

Return type

R

learn_with_optimizer(*evaluator*, *trial*, *max_epoch*=128, *validate_epoch*=5, *score_degradation_max*=5)

Learning procedures with early stopping and pruning.

Parameters

- **evaluator** (*Optional[[evaluation.Evaluator](#)]*) – The evaluator to measure the score.
- **trial** (*Optional[Trial]*) – The current optuna trial under the study (if any.)
- **max_epoch** (*int*) – Maximal number of epochs. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 128.
- **validate_epoch** (*int*) – The frequency of validation score measurement. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.
- **validate_epoch** – The frequency of validation score measurement. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.

- **score_degradation_max** (*int*) – Maximal number of allowed score degradation. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.

Return type

None

```
classmethod tune(data, evaluator, n_trials=20, timeout=None, data_suggest_function=None,
                  parameter_suggest_function=None, fixed_params={}, random_seed=None,
                  pruning_n_startup_trials=10, max_epoch=128, validate_epoch=5,
                  score_degradation_max=5, logger=None)
```

Perform the optimization step. *optuna.Study* object is created inside this function.

Parameters

- **data** (*Optional[Union[csr_matrix, csc_matrix]]*) – The training data. You can also provide tunable parameter dependent training data by providing *data_suggest_function*. In that case, data must be *None*.
- **evaluator** (*evaluation.Evaluator*) – The validation evaluator that measures the performance of the recommenders.
- **n_trials** (*int*) – The number of expected trials (including pruned ones). Defaults to 20.
- **timeout** (*Optional[int]*) – If set to some value (in seconds), the study will exit after that time period. Note that the running trials is not interrupted, though. Defaults to *None*.
- **data_suggest_function** (*Optional[Callable[[Trial], Union[csr_matrix, csc_matrix]]]*) – If not *None*, this must be a function which takes *optuna.Trial* as its argument and returns training data. Defaults to *None*.
- **parameter_suggest_function** (*Optional[Callable[[Trial], Dict[str, Any]]]*) – If not *None*, this must be a function which takes *optuna.Trial* as its argument and returns *Dict[str, Any]* (i.e., some keyword arguments of the recommender class). If *None*, *cls.default_suggest_parameter* will be used for the parameter suggestion. Defaults to *None*.
- **fixed_params** (*Dict[str, Any]*) – Fixed parameters passed to recommenders during the optimization procedure. This will replace the suggested parameter (either by *cls.default_suggest_parameter* or *parameter_suggest_function*). Defaults to *dict()*.
- **random_seed** (*Optional[int]*) – The random seed to control *optuna.samplers.TPESampler*. Defaults to *None*.
- **pruning_n_startup_trials** (*int*) – *n_startup_trials* argument passed to the constructor of *optuna.pruners.MedianPruner*.
- **max_epoch** (*int*) – The maximal number of epochs for the training. If iterative learning procedure is not available, this parameter will be ignored.
- **validate_epoch** (*int, optional*) – The frequency of validation score measurement. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.
- **score_degradation_max** (*int, optional*) – Maximal number of allowed score degradation. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5. Defaults to 5.
- **logger** (*Optional[Logger]*) –

Returns

A tuple that consists of

1. A dict containing the best paramaters. This dict can be passed to the recommender as `**kwargs`.
2. A `pandas.DataFrame` that contains the history of optimization.

Return type

Tuple[Dict[str, Any], DataFrame]

irspack.recommenders.TruncatedSVDRecommender

class `irspack.recommenders.TruncatedSVDRecommender`(*X_train_all*, *n_components*=4, *random_seed*=0)

Bases: [*BaseRecommender*](#), [*BaseRecommenderWithUserEmbedding*](#), [*BaseRecommenderWithItemEmbedding*](#)

Use (randomized) SVD to factorize the input matrix into low-rank matrices.

Parameters

- **X_train_all** (*csr_matrix*) – Input interaction matrix.
- **n_components** (*int*) – The rank of approximation. Defaults to 4. If this is larger than `X_train_all`, the value will be truncated into `X_train_all.shape[1]`
- **random_seed** (*int*) – The random seed to be passed on core TruncSVD.

__init__(*X_train_all*, *n_components*=4, *random_seed*=0)

Parameters

- **X_train_all** (*Union[csr_matrix, csc_matrix]*) –
- **n_components** (*int*) –
- **random_seed** (*int*) –

Return type

None

Methods

<code>__init__(X_train_all[, n_components, ...])</code>	
<code>default_suggest_parameter(trial, fixed_params)</code>	
<code>from_config(X_train_all, config)</code>	
<code>get_item_embedding()</code>	Get item embedding vectors.
<code>get_score(user_indices)</code>	Compute the item recommendation score for a subset of users.
<code>get_score_block(begin, end)</code>	Compute the score for a block of the users.
<code>get_score_cold_user(X)</code>	Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix.
<code>get_score_cold_user_remove_seen(X)</code>	Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix.
<code>get_score_from_item_embedding(user_indices, ...)</code>	
<code>get_score_from_user_embedding(user_embedding)</code>	Compute the item score from user embedding.
<code>get_score_remove_seen(user_indices)</code>	Compute the item score and mask the item in the training set.
<code>get_score_remove_seen_block(begin, end)</code>	Compute the score for a block of the users, and mask the items in the training set.
<code>get_user_embedding()</code>	Get user embedding vectors.
<code>learn()</code>	Learns and returns itself.
<code>learn_with_optimizer(evaluator, trial[, ...])</code>	Learning procedures with early stopping and pruning.
<code>tune(data, evaluator[, n_trials, timeout, ...])</code>	Perform the optimization step.
<code>tune_with_study(study, data, evaluator[, ...])</code>	

Attributes

<code>decomposer</code>	
<code>default_tune_range</code>	
<code>z</code>	

X_train_all: csr_matrix

The matrix to feed into recommender.

get_item_embedding()

Get item embedding vectors.

Returns

The latent vector representation of items. Its number of rows is equal to the number of the items.

Return type*ndarray***get_score(*user_indices*)**

Compute the item recommendation score for a subset of users.

Parameters

user_indices (*ndarray*) – The index defines the subset of users.

Returns

The item scores. Its shape will be (len(*user_indices*), self.n_items)

Return type*ndarray***get_score_block(*begin*, *end*)**

Compute the score for a block of the users.

Parameters

- **begin** (*int*) – where the evaluated user block begins.
- **end** (*int*) – where the evaluated user block ends.

Returns

The item scores. Its shape will be (end - begin, self.n_items)

Return type*ndarray***get_score_cold_user(*X*)**

Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix.

Parameters

X (*Union[csr_matrix, csc_matrix]*) – The profile user-item relation matrix for unseen users. Its number of rows is arbitrary, but the number of columns must be self.n_items.

Returns

Computed item scores for users. Its shape is equal to X.

Return type*ndarray***get_score_cold_user_remove_seen(*X*)**

Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix. The score will then be masked by the input.

Parameters

X (*Union[csr_matrix, csc_matrix]*) – The profile user-item relation matrix for unseen users. Its number of rows is arbitrary, but the number of columns must be self.n_items.

Returns

Computed & masked item scores for users. Its shape is equal to X.

Return type*ndarray***get_score_from_user_embedding(*user_embedding*)**

Compute the item score from user embedding. Mainly used for cold-start scenario.

Parameters

user_embedding (*ndarray*) – Latent user representation obtained elsewhere.

Returns

The score array. Its shape will be `(user_embedding.shape[0], self.n_items)`

Return type

DenseScoreArray

get_score_remove_seen(*user_indices*)

Compute the item score and mask the item in the training set. Masked items will have the score -inf.

Parameters

user_indices (*ndarray*) – Specifies the subset of users.

Returns

The masked item scores. Its shape will be `(len(user_indices), self.n_items)`

Return type

ndarray

get_score_remove_seen_block(*begin, end*)

Compute the score for a block of the users, and mask the items in the training set. Masked items will have the score -inf.

Parameters

- **begin** (*int*) – where the evaluated user block begins.
- **end** (*int*) – where the evaluated user block ends.

Returns

The masked item scores. Its shape will be `(end - begin, self.n_items)`

Return type

ndarray

get_user_embedding()

Get user embedding vectors.

Returns

The latent vector representation of users. Its number of rows is equal to the number of the users.

Return type

ndarray

learn()

Learns and returns itself.

Returns

The model after fitting process.

Parameters

self (*R*) –

Return type

R

learn_with_optimizer(*evaluator, trial, max_epoch=128, validate_epoch=5, score_degradation_max=5*)

Learning procedures with early stopping and pruning.

Parameters

- **evaluator** (*Optional[evaluation.Evaluator]*) – The evaluator to measure the score.

- **trial** (*Optional[Trial]*) – The current optuna trial under the study (if any.)
- **max_epoch** (*int*) – Maximal number of epochs. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 128.
- **validate_epoch** (*int*) – The frequency of validation score measurement. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.
- **validate_epoch** – The frequency of validation score measurement. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.
- **score_degradation_max** (*int*) – Maximal number of allowed score degradation. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.

Return type

None

classmethod tune(*data, evaluator, n_trials=20, timeout=None, data_suggest_function=None, parameter_suggest_function=None, fixed_params={}, random_seed=None, pruning_n_startup_trials=10, max_epoch=128, validate_epoch=5, score_degradation_max=5, logger=None*)

Perform the optimization step. *optuna.Study* object is created inside this function.

Parameters

- **data** (*Optional[Union[csr_matrix, csc_matrix]]*) – The training data. You can also provide tunable parameter dependent training data by providing *data_suggest_function*. In that case, data must be *None*.
- **evaluator** (*evaluation.Evaluator*) – The validation evaluator that measures the performance of the recommenders.
- **n_trials** (*int*) – The number of expected trials (including pruned ones). Defaults to 20.
- **timeout** (*Optional[int]*) – If set to some value (in seconds), the study will exit after that time period. Note that the running trials is not interrupted, though. Defaults to *None*.
- **data_suggest_function** (*Optional[Callable[[Trial], Union[csr_matrix, csc_matrix]]]*) – If not *None*, this must be a function which takes *optuna.Trial* as its argument and returns training data. Defaults to *None*.
- **parameter_suggest_function** (*Optional[Callable[[Trial], Dict[str, Any]]]*) – If not *None*, this must be a function which takes *optuna.Trial* as its argument and returns *Dict[str, Any]* (i.e., some keyword arguments of the recommender class). If *None*, *cls.default_suggest_parameter* will be used for the parameter suggestion. Defaults to *None*.
- **fixed_params** (*Dict[str, Any]*) – Fixed parameters passed to recommenders during the optimization procedure. This will replace the suggested parameter (either by *cls.default_suggest_parameter* or *parameter_suggest_function*). Defaults to *dict()*.
- **random_seed** (*Optional[int]*) – The random seed to control *optuna.samplers.TPESampler*. Defaults to *None*.
- **pruning_n_startup_trials** (*int*) – *n_startup_trials* argument passed to the constructor of *optuna.pruners.MedianPruner*.
- **max_epoch** (*int*) – The maximal number of epochs for the training. If iterative learning procedure is not available, this parameter will be ignored.
- **validate_epoch** (*int, optional*) – The frequency of validation score measurement. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.

- **score_degradation_max**(*int*, *optional*) – Maximal number of allowed score degradation. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5. Defaults to 5.
- **logger** (*Optional*[*Logger*]) –

Returns

A tuple that consists of

1. A dict containing the best paramaters. This dict can be passed to the recommender as ****kwargs**.
2. A `pandas.DataFrame` that contains the history of optimization.

Return type

`Tuple[Dict[str, Any], DataFrame]`

irspack.recommenders.CosineKNNRecommender

```
class irspack.recommenders.CosineKNNRecommender(X_train_all, shrinkage=0.0, normalize=False,
                                                  top_k=100, feature_weighting='NONE',
                                                  bm25_k1=1.2, bm25_b=0.75, n_threads=None)
```

Bases: `BaseKNNRecommender`

K-nearest neighbor recommender system based on cosine similarity. That is, the similarity matrix W is given by (column-wise top-k restricted)

$$W_{i,j} = \begin{cases} \frac{\sum_u X_{ui} X_{uj}}{\|X_{*i}\|_2 \|X_{*j}\|_2 + \text{shrinkage}} & (\text{if } \text{normalize} = \text{True}) \\ \sum_u X_{ui} X_{uj} & (\text{if } \text{normalize} = \text{False}) \end{cases}$$

Parameters

- **X_train_all** (*Union*[*scipy.sparse.csr_matrix*, *scipy.sparse.csc_matrix*]) – Input interaction matrix.
- **shrinkage** (*float*, *optional*) – The shrinkage parameter for regularization. Defaults to 0.0.
- **normalize** (*bool*, *optional*) – Whether to normalize the similarity. Defaults to False.
- **top_k** (*int*, *optional*) – Specifies the maximal number of allowed neighbors. Defaults to 100.
- **feature_weighting** (*str*, *optional*) – Specifies how to weight the feature. Must be one of:
 - "NONE" : no feature weighting
 - "TF_IDF" : TF-IDF weighting
 - "BM_25" : [Okapi BM-25 weighting](#)
 Defaults to "NONE".
- **bm25_k1** (*float*, *optional*) – The k1 parameter for BM25. Ignored if `feature_weighting` is not "BM_25". Defaults to 1.2.
- **bm25_b** (*float*, *optional*) – The b parameter for BM25. Ignored if `feature_weighting` is not "BM_25". Defaults to 0.75.

- **n_threads** (*Optional[int], optional*) – Specifies the number of threads to use for the computation. If *None*, the environment variable "IRSPACK_NUM_THREADS_DEFAULT" will be looked up, and if the variable is not set, it will be set to `os.cpu_count()`. Defaults to *None*.

__init__ (*X_train_all, shrinkage=0.0, normalize=False, top_k=100, feature_weighting='NONE', bm25_k1=1.2, bm25_b=0.75, n_threads=None*)

Parameters

- **X_train_all** (*Union[csr_matrix, csc_matrix]*) –
- **shrinkage** (*float*) –
- **normalize** (*bool*) –
- **top_k** (*int*) –
- **feature_weighting** (*str*) –
- **bm25_k1** (*float*) –
- **bm25_b** (*float*) –
- **n_threads** (*Optional[int]*) –

Return type

None

Methods

__init__ (<i>X_train_all</i> [, <i>shrinkage</i> , ...])	
default_suggest_parameter (<i>trial</i> , <i>fixed_params</i>)	
from_config (<i>X_train_all</i> , <i>config</i>)	
get_score (<i>user_indices</i>)	Compute the item recommendation score for a subset of users.
get_score_block (<i>begin</i> , <i>end</i>)	Compute the score for a block of the users.
get_score_cold_user (<i>X</i>)	Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix.
get_score_cold_user_remove_seen (<i>X</i>)	Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix.
get_score_remove_seen (<i>user_indices</i>)	Compute the item score and mask the item in the training set.
get_score_remove_seen_block (<i>begin</i> , <i>end</i>)	Compute the score for a block of the users, and mask the items in the training set.
learn ()	Learns and returns itself.
learn_with_optimizer (<i>evaluator</i> , <i>trial</i> [, ...])	Learning procedures with early stopping and pruning.
tune (<i>data</i> , <i>evaluator</i> [, <i>n_trials</i> , <i>timeout</i> , ...])	Perform the optimization step.
tune_with_study (<i>study</i> , <i>data</i> , <i>evaluator</i> [, ...])	

Attributes

<i>W</i>	The computed item-item similarity weight matrix.
<code>default_tune_range</code>	

property W: Union[csr_matrix, csc_matrix, ndarray]

The computed item-item similarity weight matrix.

X_train_all: sps.csr_matrix

The matrix to feed into recommender.

get_score(*user_indices*)

Compute the item recommendation score for a subset of users.

Parameters

user_indices (*ndarray*) – The index defines the subset of users.

Returns

The item scores. Its shape will be (len(*user_indices*), self.n_items)

Return type

ndarray

get_score_block(*begin*, *end*)

Compute the score for a block of the users.

Parameters

- **begin** (*int*) – where the evaluated user block begins.
- **end** (*int*) – where the evaluated user block ends.

Returns

The item scores. Its shape will be (end - begin, self.n_items)

Return type

ndarray

get_score_cold_user(*X*)

Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix.

Parameters

X (*Union[csr_matrix, csc_matrix]*) – The profile user-item relation matrix for unseen users. Its number of rows is arbitrary, but the number of columns must be self.n_items.

Returns

Computed item scores for users. Its shape is equal to X.

Return type

ndarray

get_score_cold_user_remove_seen(*X*)

Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix. The score will then be masked by the input.

Parameters

X (*Union[csr_matrix, csc_matrix]*) – The profile user-item relation matrix for unseen users. Its number of rows is arbitrary, but the number of columns must be self.n_items.

Returns

Computed & masked item scores for users. Its shape is equal to X.

Return type

ndarray

get_score_remove_seen(*user_indices*)

Compute the item score and mask the item in the training set. Masked items will have the score -inf.

Parameters

user_indices (*ndarray*) – Specifies the subset of users.

Returns

The masked item scores. Its shape will be (len(user_indices), self.n_items)

Return type

ndarray

get_score_remove_seen_block(*begin, end*)

Compute the score for a block of the users, and mask the items in the training set. Masked items will have the score -inf.

Parameters

- **begin** (*int*) – where the evaluated user block begins.
- **end** (*int*) – where the evaluated user block ends.

Returns

The masked item scores. Its shape will be (end - begin, self.n_items)

Return type

ndarray

learn()

Learns and returns itself.

Returns

The model after fitting process.

Parameters

self (*R*) –

Return type

R

learn_with_optimizer(*evaluator, trial, max_epoch=128, validate_epoch=5, score_degradation_max=5*)

Learning procedures with early stopping and pruning.

Parameters

- **evaluator** (*Optional[[evaluation.Evaluator](#)]*) – The evaluator to measure the score.
- **trial** (*Optional[Trial]*) – The current optuna trial under the study (if any.)
- **max_epoch** (*int*) – Maximal number of epochs. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 128.
- **validate_epoch** (*int*) – The frequency of validation score measurement. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.
- **validate_epoch** – The frequency of validation score measurement. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.

- **score_degradation_max** (*int*) – Maximal number of allowed score degradation. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.

Return type

None

```
classmethod tune(data, evaluator, n_trials=20, timeout=None, data_suggest_function=None,
                  parameter_suggest_function=None, fixed_params={}, random_seed=None,
                  pruning_n_startup_trials=10, max_epoch=128, validate_epoch=5,
                  score_degradation_max=5, logger=None)
```

Perform the optimization step. *optuna.Study* object is created inside this function.

Parameters

- **data** (*Optional[Union[csr_matrix, csc_matrix]]*) – The training data. You can also provide tunable parameter dependent training data by providing *data_suggest_function*. In that case, data must be *None*.
- **evaluator** (*evaluation.Evaluator*) – The validation evaluator that measures the performance of the recommenders.
- **n_trials** (*int*) – The number of expected trials (including pruned ones). Defaults to 20.
- **timeout** (*Optional[int]*) – If set to some value (in seconds), the study will exit after that time period. Note that the running trials is not interrupted, though. Defaults to *None*.
- **data_suggest_function** (*Optional[Callable[[Trial], Union[csr_matrix, csc_matrix]]]*) – If not *None*, this must be a function which takes *optuna.Trial* as its argument and returns training data. Defaults to *None*.
- **parameter_suggest_function** (*Optional[Callable[[Trial], Dict[str, Any]]]*) – If not *None*, this must be a function which takes *optuna.Trial* as its argument and returns *Dict[str, Any]* (i.e., some keyword arguments of the recommender class). If *None*, *cls.default_suggest_parameter* will be used for the parameter suggestion. Defaults to *None*.
- **fixed_params** (*Dict[str, Any]*) – Fixed parameters passed to recommenders during the optimization procedure. This will replace the suggested parameter (either by *cls.default_suggest_parameter* or *parameter_suggest_function*). Defaults to *dict()*.
- **random_seed** (*Optional[int]*) – The random seed to control *optuna.samplers.TPESampler*. Defaults to *None*.
- **pruning_n_startup_trials** (*int*) – *n_startup_trials* argument passed to the constructor of *optuna.pruners.MedianPruner*.
- **max_epoch** (*int*) – The maximal number of epochs for the training. If iterative learning procedure is not available, this parameter will be ignored.
- **validate_epoch** (*int, optional*) – The frequency of validation score measurement. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.
- **score_degradation_max** (*int, optional*) – Maximal number of allowed score degradation. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5. Defaults to 5.
- **logger** (*Optional[Logger]*) –

Returns

A tuple that consists of

1. A dict containing the best paramaters. This dict can be passed to the recommender as `**kwargs`.
2. A `pandas.DataFrame` that contains the history of optimization.

Return type

`Tuple[Dict[str, Any], DataFrame]`

irspack.recommenders.AsymmetricCosineKNNRecommender

```
class irspack.recommenders.AsymmetricCosineKNNRecommender(X_train_all, shrinkage=0.0, alpha=0.5,
                                                            top_k=100, feature_weighting='NONE',
                                                            bm25_k1=1.2, bm25_b=0.75,
                                                            n_threads=None)
```

Bases: `BaseKNNRecommender`

K-nearest neighbor recommender system based on asymmetric cosine similarity. That is, the similarity matrix W is given by (column-wise top-k restricted)

$$W_{i,j} = \frac{\sum_u X_{ui} X_{uj}}{\|X_{*i}\|_2^{2\alpha} \|X_{*j}\|_2^{2(1-\alpha)} + \text{shrinkage}}$$

Parameters

- **X_train_all** (`Union[scipy.sparse.csr_matrix, scipy.sparse.csc_matrix]`) – Input interaction matrix.
- **shrinkage** (`float, optional`) – The shrinkage parameter for regularization. Defaults to 0.0.
- **alpha** (`bool, optional`) – Specifies α . Defaults to 0.5.
- **top_k** (`int, optional`) – Specifies the maximal number of allowed neighbors. Defaults to 100.
- **feature_weighting** (`str, optional`) – Specifies how to weight the feature. Must be one of:
 - “NONE”: no feature weighting
 - “TF_IDF”: TF-IDF weighting
 - “BM_25”: [Okapi BM-25 weighting](#)
 Defaults to “NONE”.
- **bm25_k1** (`float, optional`) – The k_1 parameter for BM25. Ignored if `feature_weighting` is not “BM_25”. Defaults to 1.2.
- **bm25_b** (`float, optional`) – The b parameter for BM25. Ignored if `feature_weighting` is not “BM_25”. Defaults to 0.75.
- **n_threads** (`Optional[int], optional`) – Specifies the number of threads to use for the computation. If `None`, the environment variable “IRSPACK_NUM_THREADS_DEFAULT” will be looked up, and if the variable is not set, it will be set to `os.cpu_count()`. Defaults to `None`.

```
__init__(X_train_all, shrinkage=0.0, alpha=0.5, top_k=100, feature_weighting='NONE', bm25_k1=1.2,
          bm25_b=0.75, n_threads=None)
```

Parameters

- **X_train_all** (*Union[csr_matrix, csc_matrix]*) –
- **shrinkage** (*float*) –
- **alpha** (*float*) –
- **top_k** (*int*) –
- **feature_weighting** (*str*) –
- **bm25_k1** (*float*) –
- **bm25_b** (*float*) –
- **n_threads** (*Optional[int]*) –

Methods

<code>__init__(X_train_all[, shrinkage, alpha, ...])</code>	
<code>default_suggest_parameter(trial, fixed_params)</code>	
<code>from_config(X_train_all, config)</code>	
<code>get_score(user_indices)</code>	Compute the item recommendation score for a subset of users.
<code>get_score_block(begin, end)</code>	Compute the score for a block of the users.
<code>get_score_cold_user(X)</code>	Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix.
<code>get_score_cold_user_remove_seen(X)</code>	Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix.
<code>get_score_remove_seen(user_indices)</code>	Compute the item score and mask the item in the training set.
<code>get_score_remove_seen_block(begin, end)</code>	Compute the score for a block of the users, and mask the items in the training set.
<code>learn()</code>	Learns and returns itself.
<code>learn_with_optimizer(evaluator, trial[, ...])</code>	Learning procedures with early stopping and pruning.
<code>tune(data, evaluator[, n_trials, timeout, ...])</code>	Perform the optimization step.
<code>tune_with_study(study, data, evaluator[, ...])</code>	

Attributes

<code>W</code>	The computed item-item similarity weight matrix.
<code>default_tune_range</code>	

property W: `Union[csr_matrix, csc_matrix, ndarray]`

The computed item-item similarity weight matrix.

X_train_all: `sps.csr_matrix`

The matrix to feed into recommender.

get_score(*user_indices*)

Compute the item recommendation score for a subset of users.

Parameters

user_indices (*ndarray*) – The index defines the subset of users.

Returns

The item scores. Its shape will be (len(*user_indices*), self.n_items)

Return type

ndarray

get_score_block(*begin, end*)

Compute the score for a block of the users.

Parameters

- **begin** (*int*) – where the evaluated user block begins.
- **end** (*int*) – where the evaluated user block ends.

Returns

The item scores. Its shape will be (end - begin, self.n_items)

Return type

ndarray

get_score_cold_user(*X*)

Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix.

Parameters

X (*Union[csr_matrix, csc_matrix]*) – The profile user-item relation matrix for unseen users. Its number of rows is arbitrary, but the number of columns must be self.n_items.

Returns

Computed item scores for users. Its shape is equal to X.

Return type

ndarray

get_score_cold_user_remove_seen(*X*)

Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix. The score will then be masked by the input.

Parameters

X (*Union[csr_matrix, csc_matrix]*) – The profile user-item relation matrix for unseen users. Its number of rows is arbitrary, but the number of columns must be self.n_items.

Returns

Computed & masked item scores for users. Its shape is equal to X.

Return type

ndarray

get_score_remove_seen(*user_indices*)

Compute the item score and mask the item in the training set. Masked items will have the score -inf.

Parameters

user_indices (*ndarray*) – Specifies the subset of users.

Returns

The masked item scores. Its shape will be (len(user_indices), self.n_items)

Return type

ndarray

get_score_remove_seen_block(begin, end)

Compute the score for a block of the users, and mask the items in the training set. Masked items will have the score -inf.

Parameters

- **begin** (*int*) – where the evaluated user block begins.
- **end** (*int*) – where the evaluated user block ends.

Returns

The masked item scores. Its shape will be (end - begin, self.n_items)

Return type

ndarray

learn()

Learns and returns itself.

Returns

The model after fitting process.

Parameters

self (*R*) –

Return type

R

learn_with_optimizer(evaluator, trial, max_epoch=128, validate_epoch=5, score_degradation_max=5)

Learning procedures with early stopping and pruning.

Parameters

- **evaluator** (*Optional[[evaluation.Evaluator](#)]*) – The evaluator to measure the score.
- **trial** (*Optional[[Trial](#)]*) – The current optuna trial under the study (if any.)
- **max_epoch** (*int*) – Maximal number of epochs. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 128.
- **validate_epoch** (*int*) – The frequency of validation score measurement. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.
- **validate_epoch** – The frequency of validation score measurement. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.
- **score_degradation_max** (*int*) – Maximal number of allowed score degradation. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.

Return type

None

classmethod tune(*data, evaluator, n_trials=20, timeout=None, data_suggest_function=None, parameter_suggest_function=None, fixed_params={}, random_seed=None, pruning_n_startup_trials=10, max_epoch=128, validate_epoch=5, score_degradation_max=5, logger=None*)

Perform the optimization step. *optuna.Study* object is created inside this function.

Parameters

- **data** (*Optional[Union[csr_matrix, csc_matrix]]*) – The training data. You can also provide tunable parameter dependent training data by providing *data_suggest_function*. In that case, data must be *None*.
- **evaluator** (*evaluation.Evaluator*) – The validation evaluator that measures the performance of the recommenders.
- **n_trials** (*int*) – The number of expected trials (including pruned ones). Defaults to 20.
- **timeout** (*Optional[int]*) – If set to some value (in seconds), the study will exit after that time period. Note that the running trials is not interrupted, though. Defaults to *None*.
- **data_suggest_function** (*Optional[Callable[[Trial], Union[csr_matrix, csc_matrix]]]*) – If not *None*, this must be a function which takes *optuna.Trial* as its argument and returns training data. Defaults to *None*.
- **parameter_suggest_function** (*Optional[Callable[[Trial], Dict[str, Any]]]*) – If not *None*, this must be a function which takes *optuna.Trial* as its argument and returns *Dict[str, Any]* (i.e., some keyword arguments of the recommender class). If *None*, *cls.default_suggest_parameter* will be used for the parameter suggestion. Defaults to *None*.
- **fixed_params** (*Dict[str, Any]*) – Fixed parameters passed to recommenders during the optimization procedure. This will replace the suggested parameter (either by *cls.default_suggest_parameter* or *parameter_suggest_function*). Defaults to *dict()*.
- **random_seed** (*Optional[int]*) – The random seed to control *optuna.samplers.TPESampler*. Defaults to *None*.
- **pruning_n_startup_trials** (*int*) – *n_startup_trials* argument passed to the constructor of *optuna.pruners.MedianPruner*.
- **max_epoch** (*int*) – The maximal number of epochs for the training. If iterative learning procedure is not available, this parameter will be ignored.
- **validate_epoch** (*int, optional*) – The frequency of validation score measurement. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.
- **score_degradation_max** (*int, optional*) – Maximal number of allowed score degradation. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5. Defaults to 5.
- **logger** (*Optional[Logger]*) –

Returns

A tuple that consists of

1. A dict containing the best parameters. This dict can be passed to the recommender as ****kwargs**.
2. A *pandas.DataFrame* that contains the history of optimization.

Return type

Tuple[Dict[str, Any], DataFrame]

irspack.recommenders.JaccardKNNRecommender

```
class irspack.recommenders.JaccardKNNRecommender(X_train_all, shrinkage=0.0, top_k=100,  
                                                  n_threads=None)
```

Bases: BaseKNNRecommender

K-nearest neighbor recommender system based on Jaccard similarity. That is, the similarity matrix W is given by (column-wise top-k restricted)

$$W_{i,j} = \frac{|U_i \cap U_j|}{|U_i \cup U_j| + \text{shrinkage}}$$

Parameters

- ***X_train_all*** (*Union[scipy.sparse.csr_matrix, scipy.sparse.csc_matrix]*) – Input interaction matrix.
- ***shrinkage*** (*float, optional*) – The shrinkage parameter for regularization. Defaults to 0.0.
- ***top_k*** (*int, optional*) – Specifies the maximal number of allowed neighbors. Defaults to 100.
- ***n_threads*** (*Optional[int], optional*) – Specifies the number of threads to use for the computation. If None, the environment variable "IRSPACK_NUM_THREADS_DEFAULT" will be looked up, and if the variable is not set, it will be set to `os.cpu_count()`. Defaults to None.

```
__init__(X_train_all, shrinkage=0.0, top_k=100, n_threads=None)
```

Parameters

- ***X_train_all*** (*Union[csr_matrix, csc_matrix]*) –
- ***shrinkage*** (*float*) –
- ***top_k*** (*int*) –
- ***n_threads*** (*Optional[int]*) –

Return type

None

Methods

<code>__init__(X_train_all[, shrinkage, top_k, ...])</code>	
<code>default_suggest_parameter(trial, fixed_params)</code>	
<code>from_config(X_train_all, config)</code>	
<code>get_score(user_indices)</code>	Compute the item recommendation score for a subset of users.
<code>get_score_block(begin, end)</code>	Compute the score for a block of the users.
<code>get_score_cold_user(X)</code>	Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix.
<code>get_score_cold_user_remove_seen(X)</code>	Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix.
<code>get_score_remove_seen(user_indices)</code>	Compute the item score and mask the item in the training set.
<code>get_score_remove_seen_block(begin, end)</code>	Compute the score for a block of the users, and mask the items in the training set.
<code>learn()</code>	Learns and returns itself.
<code>learn_with_optimizer(evaluator, trial[, ...])</code>	Learning procedures with early stopping and pruning.
<code>tune(data, evaluator[, n_trials, timeout, ...])</code>	Perform the optimization step.
<code>tune_with_study(study, data, evaluator[, ...])</code>	

Attributes

<code>\hat{W}</code>	The computed item-item similarity weight matrix.
<code>default_tune_range</code>	

property \hat{W} : Union[csr_matrix, csc_matrix, ndarray]

The computed item-item similarity weight matrix.

$X_{\text{train_all}}$: sps.csr_matrix

The matrix to feed into recommender.

`get_score(user_indices)`

Compute the item recommendation score for a subset of users.

Parameters

`user_indices` (ndarray) – The index defines the subset of users.

Returns

The item scores. Its shape will be (len(user_indices), self.n_items)

Return type

ndarray

`get_score_block(begin, end)`

Compute the score for a block of the users.

Parameters

- **begin** (*int*) – where the evaluated user block begins.
- **end** (*int*) – where the evaluated user block ends.

Returns

The item scores. Its shape will be (end - begin, self.n_items)

Return type

ndarray

get_score_cold_user(X)

Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix.

Parameters

X (*Union[csr_matrix, csc_matrix]*) – The profile user-item relation matrix for unseen users. Its number of rows is arbitrary, but the number of columns must be self.n_items.

Returns

Computed item scores for users. Its shape is equal to X.

Return type

ndarray

get_score_cold_user_remove_seen(X)

Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix. The score will then be masked by the input.

Parameters

X (*Union[csr_matrix, csc_matrix]*) – The profile user-item relation matrix for unseen users. Its number of rows is arbitrary, but the number of columns must be self.n_items.

Returns

Computed & masked item scores for users. Its shape is equal to X.

Return type

ndarray

get_score_remove_seen(user_indices)

Compute the item score and mask the item in the training set. Masked items will have the score -inf.

Parameters

user_indices (*ndarray*) – Specifies the subset of users.

Returns

The masked item scores. Its shape will be (len(user_indices), self.n_items)

Return type

ndarray

get_score_remove_seen_block(begin, end)

Compute the score for a block of the users, and mask the items in the training set. Masked items will have the score -inf.

Parameters

- **begin** (*int*) – where the evaluated user block begins.
- **end** (*int*) – where the evaluated user block ends.

Returns

The masked item scores. Its shape will be (end - begin, self.n_items)

Return type*ndarray***learn()**

Learns and returns itself.

Returns

The model after fitting process.

Parameters

self (*R*) –

Return type*R***learn_with_optimizer**(*evaluator, trial, max_epoch=128, validate_epoch=5, score_degradation_max=5*)

Learning procedures with early stopping and pruning.

Parameters

- **evaluator** (*Optional[evaluation.Evaluator]*) – The evaluator to measure the score.
- **trial** (*Optional[Trial]*) – The current optuna trial under the study (if any.)
- **max_epoch** (*int*) – Maximal number of epochs. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 128.
- **validate_epoch** (*int*) – The frequency of validation score measurement. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.
- **validate_epoch** – The frequency of validation score measurement. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.
- **score_degradation_max** (*int*) – Maximal number of allowed score degradation. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.

Return type*None*

classmethod tune(*data, evaluator, n_trials=20, timeout=None, data_suggest_function=None, parameter_suggest_function=None, fixed_params={}, random_seed=None, pruning_n_startup_trials=10, max_epoch=128, validate_epoch=5, score_degradation_max=5, logger=None*)

Perform the optimization step. *optuna.Study* object is created inside this function.

Parameters

- **data** (*Optional[Union[csr_matrix, csc_matrix]]*) – The training data. You can also provide tunable parameter dependent training data by providing *data_suggest_function*. In that case, data must be *None*.
- **evaluator** (*evaluation.Evaluator*) – The validation evaluator that measures the performance of the recommenders.
- **n_trials** (*int*) – The number of expected trials (including pruned ones). Defaults to 20.
- **timeout** (*Optional[int]*) – If set to some value (in seconds), the study will exit after that time period. Note that the running trials is not interrupted, though. Defaults to *None*.
- **data_suggest_function** (*Optional[Callable[[Trial], Union[csr_matrix, csc_matrix]]]*) – If not *None*, this must be a function which takes *optuna.Trial* as its argument and returns training data. Defaults to *None*.

- **parameter_suggest_function** (*Optional[Callable[[Trial], Dict[str, Any]]]*) – If not *None*, this must be a function which takes *optuna.Trial* as its argument and returns *Dict[str, Any]* (i.e., some keyword arguments of the recommender class). If *None*, *cls.default_suggest_parameter* will be used for the parameter suggestion. Defaults to *None*.
- **fixed_params** (*Dict[str, Any]*) – Fixed parameters passed to recommenders during the optimization procedure. This will replace the suggested parameter (either by *cls.default_suggest_parameter* or *parameter_suggest_function*). Defaults to *dict()*.
- **random_seed** (*Optional[int]*) – The random seed to control *optuna.samplers.TPESampler*. Defaults to *None*.
- **prunning_n_startup_trials** (*int*) – *n_startup_trials* argument passed to the constructor of *optuna.pruners.MedianPruner*.
- **max_epoch** (*int*) – The maximal number of epochs for the training. If iterative learning procedure is not available, this parameter will be ignored.
- **validate_epoch** (*int, optional*) – The frequency of validation score measurement. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.
- **score_degradation_max** (*int, optional*) – Maximal number of allowed score degradation. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5. Defaults to 5.
- **logger** (*Optional[Logger]*) –

Returns

A tuple that consists of

1. A dict containing the best paramaters. This dict can be passed to the recommender as ****kwargs**.
2. A *pandas.DataFrame* that contains the history of optimization.

Return type

Tuple[Dict[str, Any], DataFrame]

irspack.recommenders.TverskyIndexKNNRecommender

```
class irspack.recommenders.TverskyIndexKNNRecommender(X_train_all, shrinkage=0.0, alpha=0.5,
                                                         beta=0.5, top_k=100, n_threads=None)
```

Bases: *BaseKNNRecommender*

K-nearest neighbor recommender system based on Tversky Index. That is, the similarity matrix *W* is given by (column-wise top-k restricted)

$$W_{i,j} = \frac{|U_i \cap U_j|}{|U_i \cap U_j| + \alpha|U_i \setminus U_j| + \beta|U_j \setminus U_i| + \text{shrinkage}}$$

Parameters

- **X_train_all** (*Union[scipy.sparse.csr_matrix, scipy.sparse.csc_matrix]*) – Input interaction matrix.
- **shrinkage** (*float, optional*) – The shrinkage parameter for regularization. Defaults to 0.0.

- **alpha** (*float, optional*) – *alpha* parameter. Defaults to 0.5.
- **beta** (*float, optional*) – *beta* parameter. Defaults to 0.5.
- **top_k** (*int, optional*) – Specifies the maximal number of allowed neighbors. Defaults to 100.
- **n_threads** (*Optional[int], optional*) – Specifies the number of threads to use for the computation. If *None*, the environment variable "IRSPACK_NUM_THREADS_DEFAULT" will be looked up, and if the variable is not set, it will be set to `os.cpu_count()`. Defaults to *None*.

`__init__(X_train_all, shrinkage=0.0, alpha=0.5, beta=0.5, top_k=100, n_threads=None)`

Parameters

- **X_train_all** (*Union[csr_matrix, csc_matrix]*) –
- **shrinkage** (*float*) –
- **alpha** (*float*) –
- **beta** (*float*) –
- **top_k** (*int*) –
- **n_threads** (*Optional[int]*) –

Return type

None

Methods

<code>__init__(X_train_all[, shrinkage, alpha, ...])</code>	
<code>default_suggest_parameter(trial, fixed_params)</code>	
<code>from_config(X_train_all, config)</code>	
<code>get_score(user_indices)</code>	Compute the item recommendation score for a subset of users.
<code>get_score_block(begin, end)</code>	Compute the score for a block of the users.
<code>get_score_cold_user(X)</code>	Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix.
<code>get_score_cold_user_remove_seen(X)</code>	Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix.
<code>get_score_remove_seen(user_indices)</code>	Compute the item score and mask the item in the training set.
<code>get_score_remove_seen_block(begin, end)</code>	Compute the score for a block of the users, and mask the items in the training set.
<code>learn()</code>	Learns and returns itself.
<code>learn_with_optimizer(evaluator, trial[, ...])</code>	Learning procedures with early stopping and pruning.
<code>tune(data, evaluator[, n_trials, timeout, ...])</code>	Perform the optimization step.
<code>tune_with_study(study, data, evaluator[, ...])</code>	

Attributes

<i>W</i>	The computed item-item similarity weight matrix.
<code>default_tune_range</code>	

property W: `Union[csr_matrix, csc_matrix, ndarray]`

The computed item-item similarity weight matrix.

X_train_all: `sps.csr_matrix`

The matrix to feed into recommender.

get_score(*user_indices*)

Compute the item recommendation score for a subset of users.

Parameters

user_indices (*ndarray*) – The index defines the subset of users.

Returns

The item scores. Its shape will be (len(user_indices), self.n_items)

Return type

ndarray

get_score_block(*begin, end*)

Compute the score for a block of the users.

Parameters

- **begin** (*int*) – where the evaluated user block begins.
- **end** (*int*) – where the evaluated user block ends.

Returns

The item scores. Its shape will be (end - begin, self.n_items)

Return type

ndarray

get_score_cold_user(*X*)

Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix.

Parameters

X (*Union[csr_matrix, csc_matrix]*) – The profile user-item relation matrix for unseen users. Its number of rows is arbitrary, but the number of columns must be self.n_items.

Returns

Computed item scores for users. Its shape is equal to X.

Return type

ndarray

get_score_cold_user_remove_seen(*X*)

Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix. The score will then be masked by the input.

Parameters

X (*Union[csr_matrix, csc_matrix]*) – The profile user-item relation matrix for unseen users. Its number of rows is arbitrary, but the number of columns must be self.n_items.

Returns

Computed & masked item scores for users. Its shape is equal to X.

Return type

ndarray

get_score_remove_seen(*user_indices*)

Compute the item score and mask the item in the training set. Masked items will have the score -inf.

Parameters

user_indices (*ndarray*) – Specifies the subset of users.

Returns

The masked item scores. Its shape will be (len(user_indices), self.n_items)

Return type

ndarray

get_score_remove_seen_block(*begin, end*)

Compute the score for a block of the users, and mask the items in the training set. Masked items will have the score -inf.

Parameters

- **begin** (*int*) – where the evaluated user block begins.
- **end** (*int*) – where the evaluated user block ends.

Returns

The masked item scores. Its shape will be (end - begin, self.n_items)

Return type

ndarray

learn()

Learns and returns itself.

Returns

The model after fitting process.

Parameters

self (*R*) –

Return type

R

learn_with_optimizer(*evaluator, trial, max_epoch=128, validate_epoch=5, score_degradation_max=5*)

Learning procedures with early stopping and pruning.

Parameters

- **evaluator** (*Optional[[evaluation.Evaluator](#)]*) – The evaluator to measure the score.
- **trial** (*Optional[Trial]*) – The current optuna trial under the study (if any.)
- **max_epoch** (*int*) – Maximal number of epochs. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 128.
- **validate_epoch** (*int*) – The frequency of validation score measurement. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.
- **validate_epoch** – The frequency of validation score measurement. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.

- **score_degradation_max** (*int*) – Maximal number of allowed score degradation. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.

Return type

None

```
classmethod tune(data, evaluator, n_trials=20, timeout=None, data_suggest_function=None,  
                 parameter_suggest_function=None, fixed_params={}, random_seed=None,  
                 pruning_n_startup_trials=10, max_epoch=128, validate_epoch=5,  
                 score_degradation_max=5, logger=None)
```

Perform the optimization step. *optuna.Study* object is created inside this function.

Parameters

- **data** (*Optional[Union[csr_matrix, csc_matrix]]*) – The training data. You can also provide tunable parameter dependent training data by providing *data_suggest_function*. In that case, data must be *None*.
- **evaluator** (*evaluation.Evaluator*) – The validation evaluator that measures the performance of the recommenders.
- **n_trials** (*int*) – The number of expected trials (including pruned ones). Defaults to 20.
- **timeout** (*Optional[int]*) – If set to some value (in seconds), the study will exit after that time period. Note that the running trials is not interrupted, though. Defaults to *None*.
- **data_suggest_function** (*Optional[Callable[[Trial], Union[csr_matrix, csc_matrix]]]*) – If not *None*, this must be a function which takes *optuna.Trial* as its argument and returns training data. Defaults to *None*.
- **parameter_suggest_function** (*Optional[Callable[[Trial], Dict[str, Any]]]*) – If not *None*, this must be a function which takes *optuna.Trial* as its argument and returns *Dict[str, Any]* (i.e., some keyword arguments of the recommender class). If *None*, *cls.default_suggest_parameter* will be used for the parameter suggestion. Defaults to *None*.
- **fixed_params** (*Dict[str, Any]*) – Fixed parameters passed to recommenders during the optimization procedure. This will replace the suggested parameter (either by *cls.default_suggest_parameter* or *parameter_suggest_function*). Defaults to *dict()*.
- **random_seed** (*Optional[int]*) – The random seed to control *optuna.samplers.TPESampler*. Defaults to *None*.
- **pruning_n_startup_trials** (*int*) – *n_startup_trials* argument passed to the constructor of *optuna.pruners.MedianPruner*.
- **max_epoch** (*int*) – The maximal number of epochs for the training. If iterative learning procedure is not available, this parameter will be ignored.
- **validate_epoch** (*int, optional*) – The frequency of validation score measurement. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.
- **score_degradation_max** (*int, optional*) – Maximal number of allowed score degradation. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5. Defaults to 5.
- **logger** (*Optional[Logger]*) –

Returns

A tuple that consists of

1. A dict containing the best paramaters. This dict can be passed to the recommender as `**kwargs`.
2. A `pandas.DataFrame` that contains the history of optimization.

Return type

`Tuple[Dict[str, Any], DataFrame]`

irspack.recommenders.CosineUserKNNRecommender

```
class irspack.recommenders.CosineUserKNNRecommender(X_train_all, shrinkage=0.0, normalize=True,
                                                    top_k=100, feature_weighting='NONE',
                                                    bm25_k1=1.2, bm25_b=0.75, n_threads=None)
```

Bases: `BaseUserKNNRecommender`

K-nearest neighbor recommender system based on cosine similarity. That is, the similarity matrix U is given by (row-wise top-k restricted)

$$U_{u,v} = \begin{cases} \frac{\sum_i X_{ui} X_{vi}}{\|X_{u*}\|_2 \|X_{v*}\|_2 + \text{shrinkage}} & (\text{if normalize} = \text{True}) \\ \sum_i X_{ui} X_{vi} & (\text{if normalize} = \text{False}) \end{cases}$$

Parameters

- **X_train_all** (`Union[scipy.sparse.csr_matrix, scipy.sparse.csc_matrix]`) – Input interaction matrix.
- **shrinkage** (`float, optional`) – The shrinkage parameter for regularization. Defaults to 0.0.
- **normalize** (`bool, optional`) – Whether to normalize the similarity. Defaults to False.
- **top_k** (`int, optional`) – Specifies the maximal number of allowed neighbors. Defaults to 100.
- **feature_weighting** (`str, optional`) – Specifies how to weight the feature. Must be one of:
 - “NONE” : no feature weighting
 - “TF_IDF” : TF-IDF weighting
 - “BM_25” : [Okapi BM-25 weighting](#)
 Defaults to “NONE”.
- **bm25_k1** (`float, optional`) – The k1 parameter for BM25. Ignored if `feature_weighting` is not “BM_25”. Defaults to 1.2.
- **bm25_b** (`float, optional`) – The b parameter for BM25. Ignored if `feature_weighting` is not “BM_25”. Defaults to 0.75.
- **n_threads** (`Optional[int], optional`) – Specifies the number of threads to use for the computation. If None, the environment variable “IRSPACK_NUM_THREADS_DEFAULT” will be looked up, and if the variable is not set, it will be set to `os.cpu_count()`. Defaults to None.

```
__init__(X_train_all, shrinkage=0.0, normalize=True, top_k=100, feature_weighting='NONE',
         bm25_k1=1.2, bm25_b=0.75, n_threads=None)
```

Parameters

- **X_train_all** (`Union[csr_matrix, csc_matrix]`) –

- `shrinkage` (*float*) –
- `normalize` (*bool*) –
- `top_k` (*int*) –
- `feature_weighting` (*str*) –
- `bm25_k1` (*float*) –
- `bm25_b` (*float*) –
- `n_threads` (*Optional[int]*) –

Methods

<code>__init__(X_train_all[, shrinkage, ...])</code>	
<code>default_suggest_parameter(trial, fixed_params)</code>	
<code>from_config(X_train_all, config)</code>	
<code>get_score(user_indices)</code>	Compute the item recommendation score for a subset of users.
<code>get_score_block(begin, end)</code>	Compute the score for a block of the users.
<code>get_score_cold_user(X)</code>	Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix.
<code>get_score_cold_user_remove_seen(X)</code>	Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix.
<code>get_score_remove_seen(user_indices)</code>	Compute the item score and mask the item in the training set.
<code>get_score_remove_seen_block(begin, end)</code>	Compute the score for a block of the users, and mask the items in the training set.
<code>learn()</code>	Learns and returns itself.
<code>learn_with_optimizer(evaluator, trial[, ...])</code>	Learning procedures with early stopping and pruning.
<code>tune(data, evaluator[, n_trials, timeout, ...])</code>	Perform the optimization step.
<code>tune_with_study(study, data, evaluator[, ...])</code>	

Attributes

<code>U</code>	The computed user-user similarity weight matrix.
<code>default_tune_range</code>	

property U: Union[csr_matrix, csc_matrix, ndarray]

The computed user-user similarity weight matrix.

X_train_all: sps.csr_matrix

The matrix to feed into recommender.

get_score(*user_indices*)

Compute the item recommendation score for a subset of users.

Parameters

user_indices (*ndarray*) – The index defines the subset of users.

Returns

The item scores. Its shape will be (len(*user_indices*), self.n_items)

Return type

ndarray

get_score_block(*begin*, *end*)

Compute the score for a block of the users.

Parameters

- **begin** (*int*) – where the evaluated user block begins.
- **end** (*int*) – where the evaluated user block ends.

Returns

The item scores. Its shape will be (end - begin, self.n_items)

Return type

ndarray

get_score_cold_user(*X*)

Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix.

Parameters

X (*Union[csr_matrix, csc_matrix]*) – The profile user-item relation matrix for unseen users. Its number of rows is arbitrary, but the number of columns must be self.n_items.

Returns

Computed item scores for users. Its shape is equal to X.

Return type

ndarray

get_score_cold_user_remove_seen(*X*)

Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix. The score will then be masked by the input.

Parameters

X (*Union[csr_matrix, csc_matrix]*) – The profile user-item relation matrix for unseen users. Its number of rows is arbitrary, but the number of columns must be self.n_items.

Returns

Computed & masked item scores for users. Its shape is equal to X.

Return type

ndarray

get_score_remove_seen(*user_indices*)

Compute the item score and mask the item in the training set. Masked items will have the score -inf.

Parameters

user_indices (*ndarray*) – Specifies the subset of users.

Returns

The masked item scores. Its shape will be (len(*user_indices*), self.n_items)

Return type*ndarray***get_score_remove_seen_block**(*begin, end*)

Compute the score for a block of the users, and mask the items in the training set. Masked items will have the score -inf.

Parameters

- **begin** (*int*) – where the evaluated user block begins.
- **end** (*int*) – where the evaluated user block ends.

Returns

The masked item scores. Its shape will be (end - begin, self.n_items)

Return type*ndarray***learn**()

Learns and returns itself.

Returns

The model after fitting process.

Parameters

self (*R*) –

Return type*R***learn_with_optimizer**(*evaluator, trial, max_epoch=128, validate_epoch=5, score_degradation_max=5*)

Learning procedures with early stopping and pruning.

Parameters

- **evaluator** (*Optional[[evaluation.Evaluator](#)]*) – The evaluator to measure the score.
- **trial** (*Optional[Trial]*) – The current optuna trial under the study (if any.)
- **max_epoch** (*int*) – Maximal number of epochs. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 128.
- **validate_epoch** (*int*) – The frequency of validation score measurement. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.
- **validate_epoch** – The frequency of validation score measurement. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.
- **score_degradation_max** (*int*) – Maximal number of allowed score degradation. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.

Return type*None*

classmethod tune(*data, evaluator, n_trials=20, timeout=None, data_suggest_function=None, parameter_suggest_function=None, fixed_params={}, random_seed=None, pruning_n_startup_trials=10, max_epoch=128, validate_epoch=5, score_degradation_max=5, logger=None*)

Perform the optimization step. *optuna.Study* object is created inside this function.

Parameters

- **data** (*Optional[Union[csr_matrix, csc_matrix]]*) – The training data. You can also provide tunable parameter dependent training data by providing *data_suggest_function*. In that case, data must be *None*.
- **evaluator** (*evaluation.Evaluator*) – The validation evaluator that measures the performance of the recommenders.
- **n_trials** (*int*) – The number of expected trials (including pruned ones). Defaults to 20.
- **timeout** (*Optional[int]*) – If set to some value (in seconds), the study will exit after that time period. Note that the running trials is not interrupted, though. Defaults to *None*.
- **data_suggest_function** (*Optional[Callable[[Trial], Union[csr_matrix, csc_matrix]]]*) – If not *None*, this must be a function which takes *optuna.Trial* as its argument and returns training data. Defaults to *None*.
- **parameter_suggest_function** (*Optional[Callable[[Trial], Dict[str, Any]]]*) – If not *None*, this must be a function which takes *optuna.Trial* as its argument and returns *Dict[str, Any]* (i.e., some keyword arguments of the recommender class). If *None*, *cls.default_suggest_parameter* will be used for the parameter suggestion. Defaults to *None*.
- **fixed_params** (*Dict[str, Any]*) – Fixed parameters passed to recommenders during the optimization procedure. This will replace the suggested parameter (either by *cls.default_suggest_parameter* or *parameter_suggest_function*). Defaults to *dict()*.
- **random_seed** (*Optional[int]*) – The random seed to control *optuna.samplers.TPESampler*. Defaults to *None*.
- **pruning_n_startup_trials** (*int*) – *n_startup_trials* argument passed to the constructor of *optuna.pruners.MedianPruner*.
- **max_epoch** (*int*) – The maximal number of epochs for the training. If iterative learning procedure is not available, this parameter will be ignored.
- **validate_epoch** (*int, optional*) – The frequency of validation score measurement. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.
- **score_degradation_max** (*int, optional*) – Maximal number of allowed score degradation. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5. Defaults to 5.
- **logger** (*Optional[Logger]*) –

Returns

A tuple that consists of

1. A dict containing the best parameters. This dict can be passed to the recommender as ****kwargs**.
2. A *pandas.DataFrame* that contains the history of optimization.

Return type

Tuple[Dict[str, Any], DataFrame]

irspack.recommenders.AsymmetricCosineUserKNNRecommender

```
class irspack.recommenders.AsymmetricCosineUserKNNRecommender(X_train_all, shrinkage=0.0,
                                                                alpha=0.5, top_k=100,
                                                                feature_weighting='NONE',
                                                                bm25_k1=1.2, bm25_b=0.75,
                                                                n_threads=None)
```

Bases: BaseUserKNNRecommender

K-nearest neighbor recommender system based on asymmetric cosine similarity. That is, the similarity matrix U is given by (row-wise top-k restricted)

$$U_{u,v} = \frac{\sum_i X_{ui} X_{vi}}{\|X_{u*}\|_2^{2\alpha} \|X_{v*}\|_2^{2(1-\alpha)} + \text{shrinkage}}$$

Parameters

- **X_train_all** (*Union[scipy.sparse.csr_matrix, scipy.sparse.csc_matrix]*) – Input interaction matrix.
- **shrinkage** (*float, optional*) – The shrinkage parameter for regularization. Defaults to 0.0.
- **alpha** (*bool, optional*) – Specifies α . Defaults to 0.5.
- **top_k** (*int, optional*) – Specifies the maximal number of allowed neighbors. Defaults to 100.
- **feature_weighting** (*str, optional*) – Specifies how to weight the feature. Must be one of:
 - "NONE": no feature weighting
 - "TF_IDF": TF-IDF weighting
 - "BM_25": [Okapi BM-25 weighting](#)
 Defaults to "NONE".
- **bm25_k1** (*float, optional*) – The k1 parameter for BM25. Ignored if feature_weighting is not "BM_25". Defaults to 1.2.
- **bm25_b** (*float, optional*) – The b parameter for BM25. Ignored if feature_weighting is not "BM_25". Defaults to 0.75.
- **n_threads** (*Optional[int], optional*) – Specifies the number of threads to use for the computation. If None, the environment variable "IRSPACK_NUM_THREADS_DEFAULT" will be looked up, and if the variable is not set, it will be set to `os.cpu_count()`. Defaults to None.

```
__init__(X_train_all, shrinkage=0.0, alpha=0.5, top_k=100, feature_weighting='NONE', bm25_k1=1.2,
          bm25_b=0.75, n_threads=None)
```

Parameters

- **X_train_all** (*Union[csr_matrix, csc_matrix]*) –
- **shrinkage** (*float*) –
- **alpha** (*float*) –
- **top_k** (*int*) –
- **feature_weighting** (*str*) –

- **bm25_k1** (*float*) –
- **bm25_b** (*float*) –
- **n_threads** (*Optional[int]*) –

Methods

<code>__init__(X_train_all[, shrinkage, alpha, ...])</code>	
<code>default_suggest_parameter(trial, fixed_params)</code>	
<code>from_config(X_train_all, config)</code>	
<code>get_score(user_indices)</code>	Compute the item recommendation score for a subset of users.
<code>get_score_block(begin, end)</code>	Compute the score for a block of the users.
<code>get_score_cold_user(X)</code>	Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix.
<code>get_score_cold_user_remove_seen(X)</code>	Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix.
<code>get_score_remove_seen(user_indices)</code>	Compute the item score and mask the item in the training set.
<code>get_score_remove_seen_block(begin, end)</code>	Compute the score for a block of the users, and mask the items in the training set.
<code>learn()</code>	Learns and returns itself.
<code>learn_with_optimizer(evaluator, trial[, ...])</code>	Learning procedures with early stopping and pruning.
<code>tune(data, evaluator[, n_trials, timeout, ...])</code>	Perform the optimization step.
<code>tune_with_study(study, data, evaluator[, ...])</code>	

Attributes

<code>U</code>	The computed user-user similarity weight matrix.
<code>default_tune_range</code>	

property U: Union[csr_matrix, csc_matrix, ndarray]

The computed user-user similarity weight matrix.

X_train_all: sps.csr_matrix

The matrix to feed into recommender.

get_score(*user_indices*)

Compute the item recommendation score for a subset of users.

Parameters

user_indices (*ndarray*) – The index defines the subset of users.

Returns

The item scores. Its shape will be (len(*user_indices*), self.n_items)

Return type*ndarray***get_score_block(begin, end)**

Compute the score for a block of the users.

Parameters

- **begin** (*int*) – where the evaluated user block begins.
- **end** (*int*) – where the evaluated user block ends.

Returns

The item scores. Its shape will be (end - begin, self.n_items)

Return type*ndarray***get_score_cold_user(X)**

Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix.

Parameters

X (*Union[csr_matrix, csc_matrix]*) – The profile user-item relation matrix for unseen users. Its number of rows is arbitrary, but the number of columns must be self.n_items.

Returns

Computed item scores for users. Its shape is equal to X.

Return type*ndarray***get_score_cold_user_remove_seen(X)**

Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix. The score will then be masked by the input.

Parameters

X (*Union[csr_matrix, csc_matrix]*) – The profile user-item relation matrix for unseen users. Its number of rows is arbitrary, but the number of columns must be self.n_items.

Returns

Computed & masked item scores for users. Its shape is equal to X.

Return type*ndarray***get_score_remove_seen(user_indices)**

Compute the item score and mask the item in the training set. Masked items will have the score -inf.

Parameters

user_indices (*ndarray*) – Specifies the subset of users.

Returns

The masked item scores. Its shape will be (len(user_indices), self.n_items)

Return type*ndarray***get_score_remove_seen_block(begin, end)**

Compute the score for a block of the users, and mask the items in the training set. Masked items will have the score -inf.

Parameters

- **begin** (*int*) – where the evaluated user block begins.
- **end** (*int*) – where the evaluated user block ends.

Returns

The masked item scores. Its shape will be (end - begin, self.n_items)

Return type

ndarray

learn()

Learns and returns itself.

Returns

The model after fitting process.

Parameters

self (*R*) –

Return type

R

learn_with_optimizer(*evaluator*, *trial*, *max_epoch*=128, *validate_epoch*=5, *score_degradation_max*=5)

Learning procedures with early stopping and pruning.

Parameters

- **evaluator** (*Optional*[[evaluation.Evaluator](#)]) – The evaluator to measure the score.
- **trial** (*Optional*[*Trial*]) – The current optuna trial under the study (if any.)
- **max_epoch** (*int*) – Maximal number of epochs. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 128.
- **validate_epoch** (*int*) – The frequency of validation score measurement. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.
- **validate_epoch** – The frequency of validation score measurement. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.
- **score_degradation_max** (*int*) – Maximal number of allowed score degradation. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.

Return type

None

classmethod tune(*data*, *evaluator*, *n_trials*=20, *timeout*=None, *data_suggest_function*=None, *parameter_suggest_function*=None, *fixed_params*={}, *random_seed*=None, *pruning_n_startup_trials*=10, *max_epoch*=128, *validate_epoch*=5, *score_degradation_max*=5, *logger*=None)

Perform the optimization step. *optuna.Study* object is created inside this function.

Parameters

- **data** (*Optional*[*Union*[*csr_matrix*, *csc_matrix*]]) – The training data. You can also provide tunable parameter dependent training data by providing *data_suggest_function*. In that case, data must be *None*.
- **evaluator** ([evaluation.Evaluator](#)) – The validation evaluator that measures the performance of the recommenders.
- **n_trials** (*int*) – The number of expected trials (including pruned ones). Defaults to 20.

- **timeout** (*Optional[int]*) – If set to some value (in seconds), the study will exit after that time period. Note that the running trials is not interrupted, though. Defaults to *None*.
- **data_suggest_function** (*Optional[Callable[[Trial], Union[csr_matrix, csc_matrix]]]*) – If not *None*, this must be a function which takes *optuna.Trial* as its argument and returns training data. Defaults to *None*.
- **parameter_suggest_function** (*Optional[Callable[[Trial], Dict[str, Any]]]*) – If not *None*, this must be a function which takes *optuna.Trial* as its argument and returns *Dict[str, Any]* (i.e., some keyword arguments of the recommender class). If *None*, *cls.default_suggest_parameter* will be used for the parameter suggestion. Defaults to *None*.
- **fixed_params** (*Dict[str, Any]*) – Fixed parameters passed to recommenders during the optimization procedure. This will replace the suggested parameter (either by *cls.default_suggest_parameter* or *parameter_suggest_function*). Defaults to *dict()*.
- **random_seed** (*Optional[int]*) – The random seed to control *optuna.samplers.TPESampler*. Defaults to *None*.
- **pruning_n_startup_trials** (*int*) – *n_startup_trials* argument passed to the constructor of *optuna.pruners.MedianPruner*.
- **max_epoch** (*int*) – The maximal number of epochs for the training. If iterative learning procedure is not available, this parameter will be ignored.
- **validate_epoch** (*int, optional*) – The frequency of validation score measurement. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.
- **score_degradation_max** (*int, optional*) – Maximal number of allowed score degradation. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5. Defaults to 5.
- **logger** (*Optional[Logger]*) –

Returns

A tuple that consists of

1. A dict containing the best paramaters. This dict can be passed to the recommender as ****kwargs**.
2. A *pandas.DataFrame* that contains the history of optimization.

Return type

Tuple[Dict[str, Any], DataFrame]

irspack.recommenders.SLIMRecommender

```
class irspack.recommenders.SLIMRecommender(X_train_all, alpha=0.05, l1_ratio=0.01, positive_only=True,
                                             n_iter=100, tol=0.0001, top_k=None, n_threads=None)
```

Bases: *BaseSimilarityRecommender*

SLIM with ElasticNet-type loss function:

$$\text{loss} = \frac{1}{2} \|X - XB\|_F^2 + \frac{\alpha(1 - l_1)U}{2} \|B\|_F^2 + \alpha l_1 U |B|$$

The implementation relies on a simple (parallelized) cyclic-coordinate descent method.

Parameters

- **X_train_all** (*csr_matrix*) – Input interaction matrix.
- **alpha** (*float*) – Determines the strength of L1/L2 regularization (see above). Defaults to 0.05.
- **l1_ratio** (*float*) – Determines the strength of L1 regularization relative to alpha. Defaults to 0.01.
- **positive_only** (*bool*) – Whether we constrain the weight matrix to be non-negative. Defaults to True.
- **n_iter** (*int*) – The number of coordinate-descent iterations. Defaults to 100.
- **tol** (*float*) – Tolerance parameter for cd iteration, i.e., if the maximal parameter change of the coordinate-descent single iteration is smaller than this value, the iteration will terminate. Defaults to 1e-4.
- **top_k** (*Optional[int]*) – Specifies the maximal number of allowed non-zero coefficients per item. Defaults to None.
- **n_threads** (*Optional[int]*) – Specifies the number of threads to use for the computation. If None, the environment variable "IRSPACK_NUM_THREADS_DEFAULT" will be looked up, and if the variable is not set, it will be set to `os.cpu_count()`. Defaults to None.

`__init__(X_train_all, alpha=0.05, l1_ratio=0.01, positive_only=True, n_iter=100, tol=0.0001, top_k=None, n_threads=None)`

Parameters

- **X_train_all** (*Union[csr_matrix, csc_matrix]*) –
- **alpha** (*float*) –
- **l1_ratio** (*float*) –
- **positive_only** (*bool*) –
- **n_iter** (*int*) –
- **tol** (*float*) –
- **top_k** (*Optional[int]*) –
- **n_threads** (*Optional[int]*) –

Methods

<code>__init__(X_train_all[, alpha, ll_ratio, ...])</code>	
<code>default_suggest_parameter(trial, fixed_params)</code>	
<code>from_config(X_train_all, config)</code>	
<code>get_score(user_indices)</code>	Compute the item recommendation score for a subset of users.
<code>get_score_block(begin, end)</code>	Compute the score for a block of the users.
<code>get_score_cold_user(X)</code>	Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix.
<code>get_score_cold_user_remove_seen(X)</code>	Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix.
<code>get_score_remove_seen(user_indices)</code>	Compute the item score and mask the item in the training set.
<code>get_score_remove_seen_block(begin, end)</code>	Compute the score for a block of the users, and mask the items in the training set.
<code>learn()</code>	Learns and returns itself.
<code>learn_with_optimizer(evaluator, trial[, ...])</code>	Learning procedures with early stopping and pruning.
<code>tune(data, evaluator[, n_trials, timeout, ...])</code>	Perform the optimization step.
<code>tune_with_study(study, data, evaluator[, ...])</code>	

Attributes

<code>\hat{W}</code>	The computed item-item similarity weight matrix.
<code>default_tune_range</code>	

property \hat{W} : Union[csr_matrix, csc_matrix, ndarray]

The computed item-item similarity weight matrix.

$X_{\text{train_all}}$: sps.csr_matrix

The matrix to feed into recommender.

`get_score(user_indices)`

Compute the item recommendation score for a subset of users.

Parameters

`user_indices` (ndarray) – The index defines the subset of users.

Returns

The item scores. Its shape will be (len(*user_indices*), self.n_items)

Return type

ndarray

`get_score_block(begin, end)`

Compute the score for a block of the users.

Parameters

- **begin** (*int*) – where the evaluated user block begins.
- **end** (*int*) – where the evaluated user block ends.

Returns

The item scores. Its shape will be (end - begin, self.n_items)

Return type

ndarray

get_score_cold_user(X)

Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix.

Parameters

X (*Union[csr_matrix, csc_matrix]*) – The profile user-item relation matrix for unseen users. Its number of rows is arbitrary, but the number of columns must be self.n_items.

Returns

Computed item scores for users. Its shape is equal to X.

Return type

ndarray

get_score_cold_user_remove_seen(X)

Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix. The score will then be masked by the input.

Parameters

X (*Union[csr_matrix, csc_matrix]*) – The profile user-item relation matrix for unseen users. Its number of rows is arbitrary, but the number of columns must be self.n_items.

Returns

Computed & masked item scores for users. Its shape is equal to X.

Return type

ndarray

get_score_remove_seen(user_indices)

Compute the item score and mask the item in the training set. Masked items will have the score -inf.

Parameters

user_indices (*ndarray*) – Specifies the subset of users.

Returns

The masked item scores. Its shape will be (len(user_indices), self.n_items)

Return type

ndarray

get_score_remove_seen_block(begin, end)

Compute the score for a block of the users, and mask the items in the training set. Masked items will have the score -inf.

Parameters

- **begin** (*int*) – where the evaluated user block begins.
- **end** (*int*) – where the evaluated user block ends.

Returns

The masked item scores. Its shape will be (end - begin, self.n_items)

Return type*ndarray***learn()**

Learns and returns itself.

Returns

The model after fitting process.

Parameters

self (*R*) –

Return type*R***learn_with_optimizer**(*evaluator, trial, max_epoch=128, validate_epoch=5, score_degradation_max=5*)

Learning procedures with early stopping and pruning.

Parameters

- **evaluator** (*Optional[[evaluation.Evaluator](#)]*) – The evaluator to measure the score.
- **trial** (*Optional[[Trial](#)]*) – The current optuna trial under the study (if any.)
- **max_epoch** (*int*) – Maximal number of epochs. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 128.
- **validate_epoch** (*int*) – The frequency of validation score measurement. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.
- **validate_epoch** – The frequency of validation score measurement. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.
- **score_degradation_max** (*int*) – Maximal number of allowed score degradation. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.

Return type*None*

classmethod tune(*data, evaluator, n_trials=20, timeout=None, data_suggest_function=None, parameter_suggest_function=None, fixed_params={}, random_seed=None, pruning_n_startup_trials=10, max_epoch=128, validate_epoch=5, score_degradation_max=5, logger=None*)

Perform the optimization step. *optuna.Study* object is created inside this function.

Parameters

- **data** (*Optional[Union[[csr_matrix](#), [csc_matrix](#)]]*) – The training data. You can also provide tunable parameter dependent training data by providing *data_suggest_function*. In that case, data must be *None*.
- **evaluator** ([evaluation.Evaluator](#)) – The validation evaluator that measures the performance of the recommenders.
- **n_trials** (*int*) – The number of expected trials (including pruned ones). Defaults to 20.
- **timeout** (*Optional[int]*) – If set to some value (in seconds), the study will exit after that time period. Note that the running trials is not interrupted, though. Defaults to *None*.
- **data_suggest_function** (*Optional[Callable[[[Trial](#)], Union[[csr_matrix](#), [csc_matrix](#)]]]*) – If not *None*, this must be a function which takes *optuna.Trial* as its argument and returns training data. Defaults to *None*.

- **parameter_suggest_function** (*Optional[Callable[[Trial], Dict[str, Any]]]*) – If not *None*, this must be a function which takes *optuna.Trial* as its argument and returns *Dict[str, Any]* (i.e., some keyword arguments of the recommender class). If *None*, *cls.default_suggest_parameter* will be used for the parameter suggestion. Defaults to *None*.
- **fixed_params** (*Dict[str, Any]*) – Fixed parameters passed to recommenders during the optimization procedure. This will replace the suggested parameter (either by *cls.default_suggest_parameter* or *parameter_suggest_function*). Defaults to *dict()*.
- **random_seed** (*Optional[int]*) – The random seed to control *optuna.samplers.TPESampler*. Defaults to *None*.
- **pruning_n_startup_trials** (*int*) – *n_startup_trials* argument passed to the constructor of *optuna.pruners.MedianPruner*.
- **max_epoch** (*int*) – The maximal number of epochs for the training. If iterative learning procedure is not available, this parameter will be ignored.
- **validate_epoch** (*int, optional*) – The frequency of validation score measurement. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.
- **score_degradation_max** (*int, optional*) – Maximal number of allowed score degradation. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5. Defaults to 5.
- **logger** (*Optional[Logger]*) –

Returns

A tuple that consists of

1. A dict containing the best paramaters. This dict can be passed to the recommender as ****kwargs**.
2. A *pandas.DataFrame* that contains the history of optimization.

Return type

Tuple[Dict[str, Any], DataFrame]

irspack.recommenders.DenseSLIMRecommender

class *irspack.recommenders.DenseSLIMRecommender*(*X_train_all, reg=1*)

Bases: *BaseSimilarityRecommender*

Implementation of DenseSLIM or Embarrassingly Shallow AutoEncoder (EASE ^R).

See:

- [Embarrassingly Shallow Autoencoders for Sparse Data](#)

Parameters

- **X_train_all** (*Union[scipy.sparse.csr_matrix, scipy.sparse.csc_matrix]*) – Input interaction matrix.
- **reg** (*float, optional*) – The regularization parameter for ease. Defaults to 1.0.

```
__init__(X_train_all, reg=1)
```

Parameters

- **X_train_all** (*Union[csr_matrix, csc_matrix]*) –
- **reg** (*float*) –

Methods

<code>__init__(X_train_all[, reg])</code>	
<code>default_suggest_parameter(trial, fixed_params)</code>	
<code>from_config(X_train_all, config)</code>	
<code>get_score(user_indices)</code>	Compute the item recommendation score for a subset of users.
<code>get_score_block(begin, end)</code>	Compute the score for a block of the users.
<code>get_score_cold_user(X)</code>	Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix.
<code>get_score_cold_user_remove_seen(X)</code>	Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix.
<code>get_score_remove_seen(user_indices)</code>	Compute the item score and mask the item in the training set.
<code>get_score_remove_seen_block(begin, end)</code>	Compute the score for a block of the users, and mask the items in the training set.
<code>learn()</code>	Learns and returns itself.
<code>learn_with_optimizer(evaluator, trial[, ...])</code>	Learning procedures with early stopping and pruning.
<code>tune(data, evaluator[, n_trials, timeout, ...])</code>	Perform the optimization step.
<code>tune_with_study(study, data, evaluator[, ...])</code>	

Attributes

<i>W</i>	The computed item-item similarity weight matrix.
<code>default_tune_range</code>	

property W: *Union[csr_matrix, csc_matrix, ndarray]*

The computed item-item similarity weight matrix.

X_train_all: *sps.csr_matrix*

The matrix to feed into recommender.

get_score(*user_indices*)

Compute the item recommendation score for a subset of users.

Parameters

user_indices (*ndarray*) – The index defines the subset of users.

Returns

The item scores. Its shape will be $(\text{len}(\text{user_indices}), \text{self.n_items})$

Return type

ndarray

get_score_block(begin, end)

Compute the score for a block of the users.

Parameters

- **begin** (*int*) – where the evaluated user block begins.
- **end** (*int*) – where the evaluated user block ends.

Returns

The item scores. Its shape will be $(\text{end} - \text{begin}, \text{self.n_items})$

Return type

ndarray

get_score_cold_user(X)

Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix.

Parameters

X (*Union[csr_matrix, csc_matrix]*) – The profile user-item relation matrix for unseen users. Its number of rows is arbitrary, but the number of columns must be `self.n_items`.

Returns

Computed item scores for users. Its shape is equal to X.

Return type

ndarray

get_score_cold_user_remove_seen(X)

Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix. The score will then be masked by the input.

Parameters

X (*Union[csr_matrix, csc_matrix]*) – The profile user-item relation matrix for unseen users. Its number of rows is arbitrary, but the number of columns must be `self.n_items`.

Returns

Computed & masked item scores for users. Its shape is equal to X.

Return type

ndarray

get_score_remove_seen(user_indices)

Compute the item score and mask the item in the training set. Masked items will have the score -inf.

Parameters

user_indices (*ndarray*) – Specifies the subset of users.

Returns

The masked item scores. Its shape will be $(\text{len}(\text{user_indices}), \text{self.n_items})$

Return type

ndarray

get_score_remove_seen_block(*begin, end*)

Compute the score for a block of the users, and mask the items in the training set. Masked items will have the score -inf.

Parameters

- **begin** (*int*) – where the evaluated user block begins.
- **end** (*int*) – where the evaluated user block ends.

Returns

The masked item scores. Its shape will be (end - begin, self.n_items)

Return type

ndarray

learn()

Learns and returns itself.

Returns

The model after fitting process.

Parameters

self (*R*) –

Return type

R

learn_with_optimizer(*evaluator, trial, max_epoch=128, validate_epoch=5, score_degradation_max=5*)

Learning procedures with early stopping and pruning.

Parameters

- **evaluator** (*Optional[[evaluation.Evaluator](#)]*) – The evaluator to measure the score.
- **trial** (*Optional[[Trial](#)]*) – The current optuna trial under the study (if any.)
- **max_epoch** (*int*) – Maximal number of epochs. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 128.
- **validate_epoch** (*int*) – The frequency of validation score measurement. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.
- **validate_epoch** – The frequency of validation score measurement. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.
- **score_degradation_max** (*int*) – Maximal number of allowed score degradation. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.

Return type

None

classmethod **tune**(*data, evaluator, n_trials=20, timeout=None, data_suggest_function=None, parameter_suggest_function=None, fixed_params={}, random_seed=None, pruning_n_startup_trials=10, max_epoch=128, validate_epoch=5, score_degradation_max=5, logger=None*)

Perform the optimization step. *optuna.Study* object is created inside this function.

Parameters

- **data** (*Optional[Union[[csr_matrix](#), [csc_matrix](#)]]*) – The training data. You can also provide tunable parameter dependent training data by providing *data_suggest_function*. In that case, data must be *None*.

- **evaluator** (`evaluation.Evaluator`) – The validation evaluator that measures the performance of the recommenders.
- **n_trials** (`int`) – The number of expected trials (including pruned ones). Defaults to 20.
- **timeout** (`Optional[int]`) – If set to some value (in seconds), the study will exit after that time period. Note that the running trials is not interrupted, though. Defaults to *None*.
- **data_suggest_function** (`Optional[Callable[[Trial], Union[csr_matrix, csc_matrix]]]`) – If not *None*, this must be a function which takes *optuna.Trial* as its argument and returns training data. Defaults to *None*.
- **parameter_suggest_function** (`Optional[Callable[[Trial], Dict[str, Any]]]`) – If not *None*, this must be a function which takes *optuna.Trial* as its argument and returns *Dict[str, Any]* (i.e., some keyword arguments of the recommender class). If *None*, *cls.default_suggest_parameter* will be used for the parameter suggestion. Defaults to *None*.
- **fixed_params** (`Dict[str, Any]`) – Fixed parameters passed to recommenders during the optimization procedure. This will replace the suggested parameter (either by *cls.default_suggest_parameter* or *parameter_suggest_function*). Defaults to *dict()*.
- **random_seed** (`Optional[int]`) – The random seed to control *optuna.samplers.TPESampler*. Defaults to *None*.
- **prunning_n_startup_trials** (`int`) – *n_startup_trials* argument passed to the constructor of *optuna.pruners.MedianPruner*.
- **max_epoch** (`int`) – The maximal number of epochs for the training. If iterative learning procedure is not available, this parameter will be ignored.
- **validate_epoch** (`int, optional`) – The frequency of validation score measurement. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.
- **score_degradation_max** (`int, optional`) – Maximal number of allowed score degradation. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5. Defaults to 5.
- **logger** (`Optional[Logger]`) –

Returns

A tuple that consists of

1. A dict containing the best paramaters. This dict can be passed to the recommender as ****kwargs**.
2. A `pandas.DataFrame` that contains the history of optimization.

Return type

`Tuple[Dict[str, Any], DataFrame]`

irspack.recommenders.get_recommender_class

irspack.recommenders.get_recommender_class(*recommender_name*)

Get recommender class from its class name.

Parameters

recommender_name (*str*) – The class name of the recommender.

Returns

The recommender class with its class name being *recommender_name*.

Return type

Type[BaseRecommender]

A LightFM wrapper for BPR matrix factorization (requires a separate installation of [lightFM](#)).

BPRFMRecommender

A [LightFM](#) wrapper for our interface.

irspack.recommenders.BPRFMRecommender

```
class irspack.recommenders.BPRFMRecommender(X_train_all, n_components=128, item_alpha=1e-09,
                                              user_alpha=1e-09, loss='bpr', n_threads=None,
                                              train_epochs=128)
```

Bases: BaseRecommenderWithEarlyStopping, BaseRecommenderWithUserEmbedding,
BaseRecommenderWithItemEmbedding

A [LightFM](#) wrapper for our interface.

This will create LightFM instance by

```
fm = LightFM(
    no_components=n_components,
    item_alpha=item_alpha,
    user_alpha=user_alpha,
    loss=loss,
)
```

and run `fm.fit_partial(X, num_threads=self.n_threads)` to train through a single epoch.

Parameters

- **X_train_all** (*csr_matrix*) – Input interaction matrix.
- **n_components** (*int*) – The dimension for latent factor. Defaults to 128.
- **item_alpha** (*float*) – The regularization coefficient for item factors. Defaults to 1e-9.
- **user_alpha** (*float*) – The regularization coefficient for user factors. Defaults to 1e-9.
- **loss** (*str*) – Specifies the loss function type of LightFM. Must be one of {"bpr", "warp"}. Defaults to "bpr".
- **train_epochs** (*int*) – Number of training epochs. Defaults to 128.
- **n_threads** (*Optional[int]*) – Specifies the number of threads to use for the computation. If None, the environment variable "IRSPACK_NUM_THREADS_DEFAULT" will be looked up, and if the variable is not set, it will be set to `os.cpu_count()`. Defaults to None.

```
__init__(X_train_all, n_components=128, item_alpha=1e-09, user_alpha=1e-09, loss='bpr',  
         n_threads=None, train_epochs=128)
```

Parameters

- **X_train_all** (*Union[csr_matrix, csc_matrix]*) –
- **n_components** (*int*) –
- **item_alpha** (*float*) –
- **user_alpha** (*float*) –
- **loss** (*str*) –
- **n_threads** (*Optional[int]*) –
- **train_epochs** (*int*) –

Methods

<code>__init__(X_train_all[, n_components, ...])</code>	
<code>default_suggest_parameter(trial, fixed_params)</code>	
<code>from_config(X_train_all, config)</code>	
<code>get_item_embedding()</code>	Get item embedding vectors.
<code>get_score(index)</code>	Compute the item recommendation score for a subset of users.
<code>get_score_block(begin, end)</code>	Compute the score for a block of the users.
<code>get_score_cold_user(X)</code>	Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix.
<code>get_score_cold_user_remove_seen(X)</code>	Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix.
<code>get_score_from_item_embedding(user_indices, ...)</code>	
<code>get_score_from_user_embedding(user_embedding)</code>	Compute the item score from user embedding.
<code>get_score_remove_seen(user_indices)</code>	Compute the item score and mask the item in the training set.
<code>get_score_remove_seen_block(begin, end)</code>	Compute the score for a block of the users, and mask the items in the training set.
<code>get_user_embedding()</code>	Get user embedding vectors.
<code>learn()</code>	Learns and returns itself.
<code>learn_with_optimizer(evaluator, trial[, ...])</code>	Learning procedures with early stopping and pruning.
<code>load_state()</code>	
<code>run_epoch()</code>	
<code>save_state()</code>	
<code>start_learning()</code>	
<code>tune(data, evaluator[, n_trials, timeout, ...])</code>	Perform the optimization step.
<code>tune_with_study(study, data, evaluator[, ...])</code>	

Attributes

<code>default_tune_range</code>
<code>fm</code>

X_train_all: `sps.csr_matrix`

The matrix to feed into recommender.

get_item_embedding()

Get item embedding vectors.

Returns

The latent vector representation of items. Its number of rows is equal to the number of the items.

Return type

ndarray

get_score(index)

Compute the item recommendation score for a subset of users.

Parameters

- **user_indices** – The index defines the subset of users.
- **index** (*ndarray*) –

Returns

The item scores. Its shape will be (len(user_indices), self.n_items)

Return type

ndarray

get_score_block(begin, end)

Compute the score for a block of the users.

Parameters

- **begin** (*int*) – where the evaluated user block begins.
- **end** (*int*) – where the evaluated user block ends.

Returns

The item scores. Its shape will be (end - begin, self.n_items)

Return type

ndarray

get_score_cold_user(X)

Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix.

Parameters

X (*Union[csr_matrix, csc_matrix]*) – The profile user-item relation matrix for unseen users. Its number of rows is arbitrary, but the number of columns must be self.n_items.

Returns

Computed item scores for users. Its shape is equal to X.

Return type

ndarray

get_score_cold_user_remove_seen(X)

Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix. The score will then be masked by the input.

Parameters

X (*Union[csr_matrix, csc_matrix]*) – The profile user-item relation matrix for unseen users. Its number of rows is arbitrary, but the number of columns must be self.n_items.

Returns

Computed & masked item scores for users. Its shape is equal to X.

Return type

ndarray

get_score_from_user_embedding(*user_embedding*)

Compute the item score from user embedding. Mainly used for cold-start scenario.

Parameters

user_embedding (*ndarray*) – Latent user representation obtained elsewhere.

Returns

The score array. Its shape will be (*user_embedding.shape[0]*, *self.n_items*)

Return type

DenseScoreArray

get_score_remove_seen(*user_indices*)

Compute the item score and mask the item in the training set. Masked items will have the score -inf.

Parameters

user_indices (*ndarray*) – Specifies the subset of users.

Returns

The masked item scores. Its shape will be (*len(user_indices)*, *self.n_items*)

Return type

ndarray

get_score_remove_seen_block(*begin*, *end*)

Compute the score for a block of the users, and mask the items in the training set. Masked items will have the score -inf.

Parameters

- **begin** (*int*) – where the evaluated user block begins.
- **end** (*int*) – where the evaluated user block ends.

Returns

The masked item scores. Its shape will be (*end - begin*, *self.n_items*)

Return type

ndarray

get_user_embedding()

Get user embedding vectors.

Returns

The latent vector representation of users. Its number of rows is equal to the number of the users.

Return type

ndarray

learn()

Learns and returns itself.

Returns

The model after fitting process.

Parameters**self** (*R*) –**Return type***R***learn_with_optimizer**(*evaluator, trial, max_epoch=128, validate_epoch=5, score_degradation_max=5*)

Learning procedures with early stopping and pruning.

Parameters

- **evaluator** (*Optional[[evaluation.Evaluator](#)]*) – The evaluator to measure the score.
- **trial** (*Optional[Trial]*) – The current optuna trial under the study (if any.)
- **max_epoch** (*int*) – Maximal number of epochs. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 128.
- **validate_epoch** (*int*) – The frequency of validation score measurement. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.
- **validate_epoch** – The frequency of validation score measurement. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.
- **score_degradation_max** (*int*) – Maximal number of allowed score degradation. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.

Return type

None

classmethod tune(*data, evaluator, n_trials=20, timeout=None, data_suggest_function=None, parameter_suggest_function=None, fixed_params={}, random_seed=None, pruning_n_startup_trials=10, max_epoch=128, validate_epoch=5, score_degradation_max=5, logger=None*)

Perform the optimization step. *optuna.Study* object is created inside this function.**Parameters**

- **data** (*Optional[Union[[csr_matrix](#), [csc_matrix](#)]]*) – The training data. You can also provide tunable parameter dependent training data by providing *data_suggest_function*. In that case, data must be *None*.
- **evaluator** ([evaluation.Evaluator](#)) – The validation evaluator that measures the performance of the recommenders.
- **n_trials** (*int*) – The number of expected trials (including pruned ones). Defaults to 20.
- **timeout** (*Optional[int]*) – If set to some value (in seconds), the study will exit after that time period. Note that the running trials is not interrupted, though. Defaults to *None*.
- **data_suggest_function** (*Optional[Callable[[[Trial](#)], Union[[csr_matrix](#), [csc_matrix](#)]]]*) – If not *None*, this must be a function which takes *optuna.Trial* as its argument and returns training data. Defaults to *None*.
- **parameter_suggest_function** (*Optional[Callable[[[Trial](#)], Dict[str, Any]]]*) – If not *None*, this must be a function which takes *optuna.Trial* as its argument and returns *Dict[str, Any]* (i.e., some keyword arguments of the recommender class). If *None*, *cls.default_suggest_parameter* will be used for the parameter suggestion. Defaults to *None*.

- **fixed_params** (*Dict[str, Any]*) – Fixed parameters passed to recommenders during the optimization procedure. This will replace the suggested parameter (either by *cls.default_suggest_parameter* or *parameter_suggest_function*). Defaults to *dict()*.
- **random_seed** (*Optional[int]*) – The random seed to control *optuna.samplers.TPESampler*. Defaults to *None*.
- **prunning_n_startup_trials** (*int*) – *n_startup_trials* argument passed to the constructor of *optuna.pruners.MedianPruner*.
- **max_epoch** (*int*) – The maximal number of epochs for the training. If iterative learning procedure is not available, this parameter will be ignored.
- **validate_epoch** (*int, optional*) – The frequency of validation score measurement. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.
- **score_degradation_max** (*int, optional*) – Maximal number of allowed score degradation. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5. Defaults to 5.
- **logger** (*Optional[Logger]*) –

Returns

A tuple that consists of

1. A dict containing the best paramaters. This dict can be passed to the recommender as ****kwargs**.
2. A *pandas.DataFrame* that contains the history of optimization.

Return type

Tuple[Dict[str, Any], DataFrame]

As a reference code based on neural networks, we have implemented a JAX version of [Mult-VAE](#), which requires *jax*, *jaxlib*, *dm-haiku*, and *optax*:

MultVAERecommender

JAX implementation of Mult-VAE, presented in "[Variational Autoencoders for Collaborative Filtering](#)".

irspack.recommenders.MultVAERecommender

```
class irspack.recommenders.MultVAERecommender(X_train_all, dim_z=16, enc_hidden_dims=256,
                                                dec_hidden_dims=None, dropout_p=0.5,
                                                l2_regularizer=0, kl_anneal_goal=0.2,
                                                anneal_end_epoch=50, minibatch_size=512,
                                                train_epochs=300, learning_rate=0.001)
```

Bases: *BaseRecommenderWithEarlyStopping*

JAX implementation of Mult-VAE, presented in "[Variational Autoencoders for Collaborative Filtering](#)".

Parameters

- **X_train_all** (*csr_matrix*) – The source data.
- **dim_z** (*int*) – The latend dimension.
- **enc_hidden_dims** (*int*) – The encoder's intermediate layer dimension.

- **dec_hidden_dims** (*Optional[int]*) – The dimensions of the decoder’s intermediate layers.
- **dropout_p** (*float*) – Dropout ratio. Defaults to 0.5.
- **l2_regularizer** (*float*) – L2 regularization coefficient. Defaults to 0.
- **kl_anneal_goal** (*float*) – beta of beta-VAE. Defaults to 0.2.
- **anneal_end_epoch** (*int*) – The epoch to complete the annealing.. Defaults to 50.
- **minibatch_size** (*int, optional*) – Minibatch size. Defaults to 512.
- **train_epochs** (*int*) – The number of epochs to run. Defaults to 300.
- **learning_rate** (*float*) – Adam optimizer’s learning rate. Defaults to 1e-3.

__init__ (*X_train_all, dim_z=16, enc_hidden_dims=256, dec_hidden_dims=None, dropout_p=0.5, l2_regularizer=0, kl_anneal_goal=0.2, anneal_end_epoch=50, minibatch_size=512, train_epochs=300, learning_rate=0.001*)

Parameters

- **X_train_all** (*Union[csr_matrix, csc_matrix]*) –
- **dim_z** (*int*) –
- **enc_hidden_dims** (*int*) –
- **dec_hidden_dims** (*Optional[int]*) –
- **dropout_p** (*float*) –
- **l2_regularizer** (*float*) –
- **kl_anneal_goal** (*float*) –
- **anneal_end_epoch** (*int*) –
- **minibatch_size** (*int*) –
- **train_epochs** (*int*) –
- **learning_rate** (*float*) –

Return type

None

Methods

<code>__init__(X_train_all[, dim_z, ...])</code>	
<code>default_suggest_parameter(trial, fixed_params)</code>	
<code>from_config(X_train_all, config)</code>	
<code>get_score(user_indices)</code>	Compute the item recommendation score for a subset of users.
<code>get_score_block(begin, end)</code>	Compute the score for a block of the users.
<code>get_score_cold_user(X)</code>	Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix.
<code>get_score_cold_user_remove_seen(X)</code>	Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix.
<code>get_score_remove_seen(user_indices)</code>	Compute the item score and mask the item in the training set.
<code>get_score_remove_seen_block(begin, end)</code>	Compute the score for a block of the users, and mask the items in the training set.
<code>learn()</code>	Learns and returns itself.
<code>learn_with_optimizer(evaluator, trial[, ...])</code>	Learning procedures with early stopping and pruning.
<code>load_state()</code>	
<code>run_epoch()</code>	
<code>save_state()</code>	
<code>start_learning()</code>	
<code>tune(data, evaluator[, n_trials, timeout, ...])</code>	Perform the optimization step.
<code>tune_with_study(study, data, evaluator[, ...])</code>	

Attributes

<code>default_tune_range</code>

X_train_all: `sps.csr_matrix`

The matrix to feed into recommender.

get_score(*user_indices*)

Compute the item recommendation score for a subset of users.

Parameters

user_indices (*ndarray*) – The index defines the subset of users.

Returns

The item scores. Its shape will be (len(*user_indices*), self.n_items)

Return type*ndarray***get_score_block(begin, end)**

Compute the score for a block of the users.

Parameters

- **begin** (*int*) – where the evaluated user block begins.
- **end** (*int*) – where the evaluated user block ends.

Returns

The item scores. Its shape will be (end - begin, self.n_items)

Return type*ndarray***get_score_cold_user(X)**

Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix.

Parameters

X (*Union[csr_matrix, csc_matrix]*) – The profile user-item relation matrix for unseen users. Its number of rows is arbitrary, but the number of columns must be self.n_items.

Returns

Computed item scores for users. Its shape is equal to X.

Return type*ndarray***get_score_cold_user_remove_seen(X)**

Compute the item recommendation score for unseen users whose profiles are given as another user-item relation matrix. The score will then be masked by the input.

Parameters

X (*Union[csr_matrix, csc_matrix]*) – The profile user-item relation matrix for unseen users. Its number of rows is arbitrary, but the number of columns must be self.n_items.

Returns

Computed & masked item scores for users. Its shape is equal to X.

Return type*ndarray***get_score_remove_seen(user_indices)**

Compute the item score and mask the item in the training set. Masked items will have the score -inf.

Parameters

user_indices (*ndarray*) – Specifies the subset of users.

Returns

The masked item scores. Its shape will be (len(user_indices), self.n_items)

Return type*ndarray***get_score_remove_seen_block(begin, end)**

Compute the score for a block of the users, and mask the items in the training set. Masked items will have the score -inf.

Parameters

- **begin** (*int*) – where the evaluated user block begins.
- **end** (*int*) – where the evaluated user block ends.

Returns

The masked item scores. Its shape will be (end - begin, self.n_items)

Return type

ndarray

learn()

Learns and returns itself.

Returns

The model after fitting process.

Parameters

self (*R*) –

Return type

R

learn_with_optimizer(*evaluator*, *trial*, *max_epoch*=128, *validate_epoch*=5, *score_degradation_max*=5)

Learning procedures with early stopping and pruning.

Parameters

- **evaluator** (*Optional*[[evaluation.Evaluator](#)]) – The evaluator to measure the score.
- **trial** (*Optional*[*Trial*]) – The current optuna trial under the study (if any.)
- **max_epoch** (*int*) – Maximal number of epochs. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 128.
- **validate_epoch** (*int*) – The frequency of validation score measurement. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.
- **validate_epoch** – The frequency of validation score measurement. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.
- **score_degradation_max** (*int*) – Maximal number of allowed score degradation. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.

Return type

None

classmethod tune(*data*, *evaluator*, *n_trials*=20, *timeout*=None, *data_suggest_function*=None, *parameter_suggest_function*=None, *fixed_params*={}, *random_seed*=None, *pruning_n_startup_trials*=10, *max_epoch*=128, *validate_epoch*=5, *score_degradation_max*=5, *logger*=None)

Perform the optimization step. *optuna.Study* object is created inside this function.

Parameters

- **data** (*Optional*[*Union*[*csr_matrix*, *csc_matrix*]]) – The training data. You can also provide tunable parameter dependent training data by providing *data_suggest_function*. In that case, data must be *None*.
- **evaluator** ([evaluation.Evaluator](#)) – The validation evaluator that measures the performance of the recommenders.
- **n_trials** (*int*) – The number of expected trials (including pruned ones). Defaults to 20.

- **timeout** (*Optional[int]*) – If set to some value (in seconds), the study will exit after that time period. Note that the running trials is not interrupted, though. Defaults to *None*.
- **data_suggest_function** (*Optional[Callable[[Trial], Union[csr_matrix, csc_matrix]]]*) – If not *None*, this must be a function which takes *optuna.Trial* as its argument and returns training data. Defaults to *None*.
- **parameter_suggest_function** (*Optional[Callable[[Trial], Dict[str, Any]]]*) – If not *None*, this must be a function which takes *optuna.Trial* as its argument and returns *Dict[str, Any]* (i.e., some keyword arguments of the recommender class). If *None*, *cls.default_suggest_parameter* will be used for the parameter suggestion. Defaults to *None*.
- **fixed_params** (*Dict[str, Any]*) – Fixed parameters passed to recommenders during the optimization procedure. This will replace the suggested parameter (either by *cls.default_suggest_parameter* or *parameter_suggest_function*). Defaults to *dict()*.
- **random_seed** (*Optional[int]*) – The random seed to control *optuna.samplers.TPESampler*. Defaults to *None*.
- **prunning_n_startup_trials** (*int*) – *n_startup_trials* argument passed to the constructor of *optuna.pruners.MedianPruner*.
- **max_epoch** (*int*) – The maximal number of epochs for the training. If iterative learning procedure is not available, this parameter will be ignored.
- **validate_epoch** (*int, optional*) – The frequency of validation score measurement. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5.
- **score_degradation_max** (*int, optional*) – Maximal number of allowed score degradation. If iterative learning procedure is not available, this parameter will be ignored. Defaults to 5. Defaults to 5.
- **logger** (*Optional[Logger]*) –

Returns

A tuple that consists of

1. A dict containing the best paramaters. This dict can be passed to the recommender as ****kwargs**.
2. A *pandas.DataFrame* that contains the history of optimization.

Return type

Tuple[Dict[str, Any], DataFrame]

1.4.3 Split Functions

<i>UserTrainTestInteractionPair</i>	A class to hold users' train & test (if any) interactions and their ids.
<i>rowwise_train_test_split</i>	Splits the non-zero elements of a sparse matrix into two (train & test interactions).
<i>split_dataframe_partial_user_holdout</i>	Splits the DataFrame and build an interaction matrix, holding out random interactions for a subset of randomly selected users (whom we call "validation users" and "test users").
<i>holdout_specific_interactions</i>	Holds-out (part of) the interactions specified by the users.
<i>split_last_n_interaction_df</i>	Split a dataframe holding out last <i>n_holdout</i> or last <i>held-out_ratio</i> part of interactions of the users.

irspack.split.UserTrainTestInteractionPair

class irspack.split.UserTrainTestInteractionPair(*user_ids*, *X_train*, *X_test*, *item_ids=None*)

Bases: object

A class to hold users' train & test (if any) interactions and their ids.

Parameters

- **user_ids** (*Union[List[Any], ndarray]*) – List of user ids. Its *i*-th element should correspond to *i*-th row of *X_train*.
- **X_train** (*csr_matrix*) – The train part of interactions.
- **X_test** (*csr_matrix*) – The test part of interactions (if any). If *None*, an empty matrix with the shape of *X_train* will be created. Defaults to *None*.
- **item_ids** (*Optional[Union[List[Any], ndarray]]*) –

Raises

- **ValueError** – when *X_train* and *user_ids* have inconsistent size.
- **ValueError** – when *X_train* and *X_test* have inconsistent size.

__init__(*user_ids*, *X_train*, *X_test*, *item_ids=None*)

Parameters

- **user_ids** (*Union[List[Any], ndarray]*) –
- **X_train** (*Union[csr_matrix, csc_matrix]*) –
- **X_test** (*Optional[Union[csr_matrix, csc_matrix]]*) –
- **item_ids** (*Optional[Union[List[Any], ndarray]]*) –

Methods

<code>__init__(user_ids, X_train, X_test[, item_ids])</code>	
<code>concat(other)</code>	Concatenate the users data.
<code>df_test()</code>	
<code>df_train()</code>	

Attributes

<code>X_train</code>	The train part of users' interactions.
<code>X_test</code>	The test part of users' interactions.
<code>n_users</code>	The number of users
<code>n_items</code>	The number of items
<code>X_all</code>	If <code>X_test</code> is not None, equal to <code>X_train + X_test</code> . Otherwise equals <code>X_train</code> .

X_all: csr_matrix

If `X_test` is not None, equal to `X_train + X_test`. Otherwise equals `X_train`.

X_test: csr_matrix

The test part of users' interactions.

X_train: csr_matrix

The train part of users' interactions.

concat(*other*)

Concatenate the users data. `user_id` will be `self.user_ids + self.item_ids`.

Returns

[description]

Return type

[type]

Parameters

other (`UserTrainTestInteractionPair`) –

ValueError:

when `self` and `other` have unequal `n_items`.

n_items: int

The number of items

n_users: int

The number of users

irspack.split.rowwise_train_test_split

```
irspack.split.rowwise_train_test_split(X, test_ratio=0.5, n_test=None, ceil_n_heldout=False,
                                       random_state=None)
```

Splits the non-zero elements of a sparse matrix into two (train & test interactions). For each row, the ratio of non-zero elements that become the test interaction is (approximately) constant.

Parameters

- **X** (*Union[csr_matrix, csc_matrix]*) – The source sparse matrix.
- **test_ratio** (*float*) – The ratio of test interactions for each row. That is, for each row, if it contains NNZ-nonzero elements, the number of elements entering into the test interaction will be `math.floor(test_ratio * NNZ)`. Defaults to 0.5.
- **random_state** (*Union[None, int, RandomState]*) – The random state. Defaults to *None*.
- **n_test** (*Optional[int]*) –
- **ceil_n_heldout** (*bool*) –

Returns

A tuple of train & test interactions, which sum back to the original matrix.

Return type

Tuple[Union[csr_matrix, csc_matrix], Union[csr_matrix, csc_matrix]]

irspack.split.split_dataframe_partial_user_holdout

```
irspack.split.split_dataframe_partial_user_holdout(df_all, user_column, item_column,
                                                  time_column=None, rating_column=None,
                                                  n_val_user=None, n_test_user=None,
                                                  val_user_ratio=0.1, test_user_ratio=0.1,
                                                  heldout_ratio_val=0.5, n_heldout_val=None,
                                                  heldout_ratio_test=0.5, n_heldout_test=None,
                                                  ceil_n_heldout=False, random_state=None)
```

Splits the DataFrame and build an interaction matrix, holding out random interactions for a subset of randomly selected users (whom we call “validation users” and “test users”).

Parameters

- **df_all** (*DataFrame*) – The user-item interaction event log.
- **user_column** (*str*) – The column name for user_id.
- **item_column** (*str*) – The column name for movie_id.
- **time_column** (*Optional[str]*) – The column name (if any) specifying the time of the interaction. If this is set, the split will be based on time, and some of the most recent interactions will be held out for each user. Defaults to *None*.
- **rating_column** (*Optional[str]*) – The column name for ratings. If *None*, the rating will be treated as 1 for all interactions. Defaults to *None*.
- **n_val_user** (*Optional[int]*) – The number of “validation users”. Defaults to *None*.
- **n_test_user** (*Optional[int]*) – The number of “test users”. Defaults to *None*.
- **val_user_ratio** (*float*) – The percentage of “validation users” with respect to all users. Ignored when *n_val_user* is set. Defaults to 0.1.

- **test_user_ratio** (*float*) – The percentage of “test users” with respect to all users. Ignored when `n_text_user` is set. Defaults to 0.1.
- **heldout_ratio_val** (*float*) – The percentage of held-out interactions for “validation users”. Ignored if `n_heldout_val` is specified. Defaults to 0.5.
- **n_heldout_val** (*Optional[int]*) – The maximal number of held-out interactions for “validation users”.
- **heldout_ratio_test** (*float*) – The percentage of held-out interactions for “test users”. Ignored if `n_heldout_test` is specified. Defaults to 0.5.
- **n_heldout_val** – The maximal number of held-out interactions for “test users”.
- **ceil_n_heldout** (*bool*) – If *True*, the number of held-out interactions of user *u* will be $\text{ceil}(\text{heldout_ratio_val} * N_u)$ and $\text{ceil}(\text{heldout_ratio_test} * N_u)$. If *False*, *floor* function will be used instead. Defaults to *False*.
- **random_state** (*Union[None, int, RandomState]*) – The random state for this procedure. Defaults to *None*.
- **n_heldout_test** (*Optional[int]*) –

Raises

ValueError – When `n_val_user + n_test_user` is greater than the number of total users.

Returns

1. A dictionary with "train", "val", "test" as its keys and the corresponding dataset as its values.
2. List of unique item ids (which corresponds to the columns of the datasets).

Return type

A tuple consisting of

irspack.split.holdout_specific_interactions

```
irspack.split.holdout_specific_interactions(df, user_column, item_column, interaction_indicator,
                                           validatable_user_ratio_val=0.2,
                                           validatable_user_ratio_test=0.2, random_state=None)
```

Holds-out (part of) the interactions specified by the users.

All the users will be split into two category:

1. Those who have an interaction in the specified subset. We denote them as “validatable” users.
2. Those who don’t.

We split the users in 1. into three parts (train, validation, test)-users, and hold-out the specified interactions. The interactions of non-validatable users will be part of the train dataset.

This split will be useful when want to:

- recommend only part of the items (e.g., rather unpopular ones) to the users. In this case, the held-out interactions will be the ones with these specific items.
- split the dataframe by a certain timepoint, and ensure that no information after that timepoint contaminates the training set.

Parameters

- **df** (*DataFrame*) – The data source.

- **user_column** (*str*) – The column name of the users.
- **item_column** (*str*) – The column name of the items.
- **interaction_indicator** (*ndarray*) – Specifies where in *df* the held-out interactions are.
- **validatable_user_ratio_val** (*float*) – The ratio of “validation-set users” in the “validatable users”. Defaults to 0.2.
- **validatable_user_ratio_test** – The ratio of “test-set users” in the “validatable users”. Defaults to 0.2.
- **random_state** (*Union[None, int, RandomState]*) – The random seed used to split validatable users into three. Defaults to *None*.
- **validatable_user_ratio_test** (*float*) –

Returns

A tuple consisting of

- The aligned list of all the items.
- A dictionary with train/val/test user pairs.

Return type

Tuple[List[Any], Dict[str, UserTrainTestInteractionPair]]

irspack.split.split_last_n_interaction_df

`irspack.split.split_last_n_interaction_df(df, user_column, timestamp_column, n_heldout=None, heldout_ratio=0.1, ceil_n_heldout=False)`

Split a dataframe holding out last *n_heldout* or last *heldout_ratio* part of interactions of the users.

Parameters

- **df** (*DataFrame*) – The Dataframe to be split.
- **user_column** (*str*) – The column name for users.
- **timestamp_column** (*str*) – The column name for “timestamp” (it doesn’t have to be date-time).
- **n_heldout** (*Optional[int]*) – If not *None*, specifies the maximal number of last actions to be held-out. Defaults to *None*.
- **heldout_ratio** (*float*) – Specifies how much of each user interaction will be held out. Ignored if *n_heldout* is present.
- **ceil_n_heldout** (*bool*) – If this is *True* and *n_heldout* is *None*, the number of test interaction for a given user *u* will be $\text{ceil}(N_u * \text{heldout_ratio})$ where N_u is the number of interactions for *u*. If this is *False*, $\text{floor}(N_u * \text{heldout_ratio})$ will be used instead. Defaults to *False*.

Returns

First interactions and held-out interactions.

Return type

Tuple[DataFrame, DataFrame]

1.4.4 Utilities

<i>ItemIDMapper</i>	A utility class that helps mapping item IDs to indices or vice versa.
<i>IDMapper</i>	A utility class that helps mapping user/item IDs to indices or vice versa.

irspack.utils.ItemIDMapper

class irspack.utils.ItemIDMapper(*item_ids*)

Bases: Generic[ItemIdType]

A utility class that helps mapping item IDs to indices or vice versa.

Parameters

item_ids – List of item IDs. The ordering of this list should be consistent with the item indices of recommenders or score arrays to be used.

Raises

ValueError – When there is a duplicate in item_ids.

__init__(*item_ids*)

Parameters

item_ids (*List[ItemIdType]*) –

Methods

__init__ (<i>item_ids</i>)	
<i>list_of_user_profile_to_matrix</i> (<i>users_info</i>)	Converts users' profiles (interaction histories for the users) into a sparse matrix.
<i>recommend_for_new_user</i> (<i>recommender</i> , <i>user_profile</i>)	Retrieves recommendations for an unknown user by using the user's contact history with the known items. :param recommender: The recommender for scoring. :param user_profile: User's profile given either as a list of item ids the user had a contact or a item id-rating dict. Previously unseen item ID will be ignored. :param cutoff: Maximal number of recommendations allowed. :param allowed_item_ids: If not None, recommend the items within this list. If None, all known item ids can be recommended (except for those in <i>item_ids</i> argument). Defaults to None. :param forbidden_item_ids: If not None, never recommend the items within the list. Defaults to None.
<i>recommend_for_new_user_batch</i> (<i>recommender</i> , ...)	Retrieves recommendations for unknown users by using their contact history with the known items.
<i>score_to_recommended_items</i> (<i>score</i> , <i>cutoff</i> [, ...])	
<i>score_to_recommended_items_batch</i> (<i>score</i> , <i>cutoff</i>)	Retrieve recommendation from score array.

list_of_user_profile_to_matrix(*users_info*)

Converts users' profiles (interaction histories for the users) into a sparse matrix.

Parameters

users_info (*Sequence[Union[List[ItemIdType], Dict[ItemIdType, float]]]*) – A list of user profiles. Each profile should be either the item ids that the user contacted or a dictionary of item ratings. Previously unseen item IDs will be ignored.

Returns

The converted sparse matrix. Each column corresponds to *self.items_ids*.

Return type

csr_matrix

recommend_for_new_user(*recommender, user_profile, cutoff=20, allowed_item_ids=None, forbidden_item_ids=None*)

Retrieves recommendations for an unknown user by using the user's contact history with the known items.
:param recommender: The recommender for scoring. :param user_profile: User's profile given either as a list of item ids the user had a contact or a item id-rating dict.

Previously unseen item ID will be ignored.

Parameters

- **cutoff** (*int*) – Maximal number of recommendations allowed.
- **allowed_item_ids** (*Optional[List[ItemIdType]]*) – If not *None*, recommend the items within this list. If *None*, all known item ids can be recommended (except for those in *item_ids* argument). Defaults to *None*.
- **forbidden_item_ids** (*Optional[List[ItemIdType]]*) – If not *None*, never recommend the items within the list. Defaults to *None*.
- **recommender** (*BaseRecommender*) –
- **user_profile** (*Union[List[ItemIdType], Dict[ItemIdType, float]]*) –

Returns

A List of tuples consisting of (*item_id*, *score*).

Return type

List[Tuple[ItemIdType, float]]

recommend_for_new_user_batch(*recommender, user_profiles, cutoff=20, allowed_item_ids=None, per_user_allowed_item_ids=None, forbidden_item_ids=None, n_threads=None*)

Retrieves recommendations for unknown users by using their contact history with the known items.

Parameters

- **recommender** (*BaseRecommender*) – The recommender for scoring.
- **user_profiles** (*Sequence[Union[List[ItemIdType], Dict[ItemIdType, float]]]*) – A list of user profiles. Each profile should be either the item ids the user had a contact, or item-rating dict. Previously unseen item IDs will be ignored.
- **cutoff** (*int*) – Maximal number of recommendations allowed.
- **allowed_item_ids** (*Optional[List[ItemIdType]]*) – If not *None*, defines “a list of recommendable item IDs”. Ignored if *per_user_allowed_item_ids* is set.

- **per_user_allowed_item_ids** (*Optional[List[List[ItemIdType]]]*) – If not None, defines “a list of list of recommendable item IDs” and `len(allowed_item_ids)` must be equal to `score.shape[0]`. Defaults to None.
- **forbidden_item_ids** (*Optional[List[List[ItemIdType]]]*) – If not None, defines “a list of list of forbidden item IDs” and `len(allowed_item_ids)` must be equal to `len(item_ids)`. Defaults to None.
- **n_threads** (*Optional[int]*) – Specifies the number of threads to use for the computation. If None, the environment variable "IRSPACK_NUM_THREADS_DEFAULT" will be looked up, and if the variable is not set, it will be set to `os.cpu_count()`. Defaults to None.

Returns

A list of list of tuples consisting of (`item_id`, `score`). Each internal list corresponds to the recommender’s recommendation output.

Return type

`List[List[Tuple[ItemIdType, float]]]`

score_to_recommended_items_batch(*score*, *cutoff*, *allowed_item_ids=None*,
per_user_allowed_item_ids=None, *forbidden_item_ids=None*,
n_threads=None)

Retrieve recommendation from score array. An item with negative infinity score for a user will not be recommended for the user.

Parameters

- **score** (*ndarray*) – 1d numpy ndarray for score.
- **cutoff** (*int*) – Maximal number of recommendations allowed.
- **allowed_item_ids** (*Optional[List[ItemIdType]]*) – If not None, defines “a list of recommendable item IDs”. Ignored if *per_user_allowed_item_ids* is set.
- **per_user_allowed_item_ids** (*Optional[List[List[ItemIdType]]]*) – If not None, defines “a list of list of recommendable item IDs” and `len(allowed_item_ids)` must be equal to `score.shape[0]`. Defaults to None.
- **allowed_item_ids** – If not None, defines “a list of list of recommendable item IDs” and `len(allowed_item_ids)` must be equal to `len(item_ids)`. Defaults to None.
- **forbidden_item_ids** (*Optional[List[List[ItemIdType]]]*) – If not None, defines “a list of list of forbidden item IDs” and `len(allowed_item_ids)` must be equal to `len(item_ids)`. Defaults to None.
- **n_threads** (*Optional[int]*) –

Return type

`List[List[Tuple[ItemIdType, float]]]`

irspack.utils.IDMapper

class irspack.utils.IDMapper(*user_ids*, *item_ids*)

Bases: Generic[UserIdType, ItemIdType], [ItemIDMapper](#)[ItemIdType]

A utility class that helps mapping user/item IDs to indices or vice versa.

Parameters

- **user_ids** – List of user IDs. The ordering should be consistent with the user indices of recommenders to be used.
- **item_ids** – List of item IDs. The ordering should be consistent with the item indices of recommenders or score arrays to be used.

Raises

ValueError – When there is a duplicate in *item_ids*.

__init__(*user_ids*, *item_ids*)

Parameters

- **user_ids** (*List*[*UserIdType*]) –
- **item_ids** (*List*[*ItemIdType*]) –

Methods

<code>__init__(user_ids, item_ids)</code>	
<code>list_of_user_profile_to_matrix(users_info)</code>	Converts users' profiles (interaction histories for the users) into a sparse matrix.
<code>recommend_for_known_user_batch(recommender, ...)</code>	Retrieves recommendation for known users.
<code>recommend_for_known_user_id(recommender, user_id)</code>	Retrieve recommendation result for a known user. :param recommender: The recommender for scoring. :param user_id: The target user ID. :param cutoff: Maximal number of recommendations allowed. :param allowed_item_ids: If not <code>None</code> , recommend the items within this list. If <code>None</code> , all known item ids can be recommended (except for those in <code>item_ids</code> argument). Defaults to <code>None</code> . :param forbidden_item_ids: If not <code>None</code> , never recommend the items within the list. Defaults to <code>None</code> .
<code>recommend_for_new_user(recommender, user_profile)</code>	Retrieves recommendations for an unknown user by using the user's contact history with the known items. :param recommender: The recommender for scoring. :param user_profile: User's profile given either as a list of item ids the user had a contact or a item id-rating dict. Previously unseen item ID will be ignored. :param cutoff: Maximal number of recommendations allowed. :param allowed_item_ids: If not <code>None</code> , recommend the items within this list. If <code>None</code> , all known item ids can be recommended (except for those in <code>item_ids</code> argument). Defaults to <code>None</code> . :param forbidden_item_ids: If not <code>None</code> , never recommend the items within the list. Defaults to <code>None</code> .
<code>recommend_for_new_user_batch(recommender, ...)</code>	Retrieves recommendations for unknown users by using their contact history with the known items.
<code>score_to_recommended_items(score, cutoff[, ...])</code>	
<code>score_to_recommended_items_batch(score, cutoff)</code>	Retrieve recommendation from score array.

`list_of_user_profile_to_matrix(users_info)`

Converts users' profiles (interaction histories for the users) into a sparse matrix.

Parameters

users_info (`Sequence[Union[List[ItemIdType], Dict[ItemIdType, float]]]`) – A list of user profiles. Each profile should be either the item ids that the user contacted or a dictionary of item ratings. Previously unseen item IDs will be ignored.

Returns

The converted sparse matrix. Each column correspond to `self.items_ids`.

Return type

`csr_matrix`

`recommend_for_known_user_batch(recommender, user_ids, cutoff=20, allowed_item_ids=None, per_user_allowed_item_ids=None, forbidden_item_ids=None, n_threads=None)`

Retrieves recommendation for known users.

Parameters

- **recommender** ([BaseRecommender](#)) – The recommender for scoring.
- **user_ids** (*List*[*UserIdType*]) – A list of user ids.
- **cutoff** (*int*) – Maximal number of recommendations allowed.
- **allowed_item_ids** (*Optional*[*List*[*ItemIdType*]]) – If not *None*, defines “a list of recommendable item IDs”. Ignored if *per_user_allowed_item_ids* is set.
- **per_user_allowed_item_ids** (*Optional*[*List*[*List*[*ItemIdType*]]]) – If not *None*, defines “a list of list of recommendable item IDs” and `len(allowed_item_ids)` must be equal to `score.shape[0]`. Defaults to *None*.
- **forbidden_item_ids** (*Optional*[*List*[*List*[*ItemIdType*]]]) – If not *None*, defines “a list of list of forbidden item IDs” and `len(allowed_item_ids)` must be equal to `len(item_ids)`. Defaults to *None*.
- **n_threads** (*Optional*[*int*]) – Specifies the number of threads to use for the computation. If *None*, the environment variable “IRSPACK_NUM_THREADS_DEFAULT” will be looked up, and if the variable is not set, it will be set to `os.cpu_count()`. Defaults to *None*.

Returns

A list of list of tuples consisting of (*item_id*, *score*). Each internal list corresponds to the recommender’s recommendation output.

Return type

List[*List*[*Tuple*[*ItemIdType*, *float*]]]

recommend_for_known_user_id(*recommender*, *user_id*, *cutoff*=20, *allowed_item_ids*=*None*, *forbidden_item_ids*=*None*)

Retrieve recommendation result for a known user. :param *recommender*: The recommender for scoring. :param *user_id*: The target user ID. :param *cutoff*: Maximal number of recommendations allowed. :param *allowed_item_ids*: If not *None*, recommend the items within this list.

If *None*, all known item ids can be recommended (except for those in *item_ids* argument). Defaults to *None*.

Parameters

- **forbidden_item_ids** (*Optional*[*List*[*ItemIdType*]]) – If not *None*, never recommend the items within the list. Defaults to *None*.
- **recommender** ([BaseRecommender](#)) –
- **user_id** (*UserIdType*) –
- **cutoff** (*int*) –
- **allowed_item_ids** (*Optional*[*List*[*ItemIdType*]]) –

Raises

RuntimeError – When *user_id* is not in `self.user_ids`.

Returns

A List of tuples consisting of (*item_id*, *score*).

Return type

List[*Tuple*[*ItemIdType*, *float*]]

recommend_for_new_user(*recommender*, *user_profile*, *cutoff*=20, *allowed_item_ids*=None, *forbidden_item_ids*=None)

Retrieves recommendations for an unknown user by using the user's contact history with the known items.
:param recommender: The recommender for scoring. :param user_profile: User's profile given either as a list of item ids the user had a cotact or a item id-rating dict.

Previously unseen item ID will be ignored.

Parameters

- **cutoff** (*int*) – Maximal number of recommendations allowed.
- **allowed_item_ids** (*Optional[List[ItemIdType]]*) – If not None, recommend the items within this list. If None, all known item ids can be recommended (except for those in *item_ids* argument). Defaults to None.
- **forbidden_item_ids** (*Optional[List[ItemIdType]]*) – If not None, never recommend the items within the list. Defaults to None.
- **recommender** (*BaseRecommender*) –
- **user_profile** (*Union[List[ItemIdType], Dict[ItemIdType, float]]*) –

Returns

A List of tuples consisting of (*item_id*, *score*).

Return type

List[Tuple[ItemIdType, float]]

recommend_for_new_user_batch(*recommender*, *user_profiles*, *cutoff*=20, *allowed_item_ids*=None, *per_user_allowed_item_ids*=None, *forbidden_item_ids*=None, *n_threads*=None)

Retrieves recommendations for unknown users by using their contact history with the known items.

Parameters

- **recommender** (*BaseRecommender*) – The recommender for scoring.
- **user_profiles** (*Sequence[Union[List[ItemIdType], Dict[ItemIdType, float]]]*) – A list of user profiles. Each profile should be either the item ids the user had a cotact, or item-rating dict. Previously unseen item IDs will be ignored.
- **cutoff** (*int*) – Maximal number of recommendations allowed.
- **allowed_item_ids** (*Optional[List[ItemIdType]]*) – If not None, defines “a list of recommendable item IDs”. Ignored if *per_user_allowed_item_ids* is set.
- **per_user_allowed_item_ids** (*Optional[List[List[ItemIdType]]]*) – If not None, defines “a list of list of recommendable item IDs” and `len(allowed_item_ids)` must be equal to `score.shape[0]`. Defaults to None.
- **forbidden_item_ids** (*Optional[List[List[ItemIdType]]]*) – If not None, defines “a list of list of forbidden item IDs” and `len(allowed_item_ids)` must be equal to `len(item_ids)` Defaults to None.
- **n_threads** (*Optional[int]*) – Specifies the number of threads to use for the computation. If None, the environment variable "IRSPACK_NUM_THREADS_DEFAULT" will be looked up, and if the variable is not set, it will be set to `os.cpu_count()`. Defaults to None.

Returns

A list of list of tuples consisting of (`item_id`, `score`). Each internal list corresponds to the recommender's recommendation output.

Return type

List[List[Tuple[ItemIdType, float]]]

score_to_recommended_items_batch(*score*, *cutoff*, *allowed_item_ids=None*,
per_user_allowed_item_ids=None, *forbidden_item_ids=None*,
n_threads=None)

Retrieve recommendation from score array. An item with negative infinity score for a user will not be recommended for the user.

Parameters

- **score** (*ndarray*) – 1d numpy ndarray for score.
- **cutoff** (*int*) – Maximal number of recommendations allowed.
- **allowed_item_ids** (*Optional[List[ItemIdType]]*) – If not `None`, defines “a list of recommendable item IDs”. Ignored if *per_user_allowed_item_ids* is set.
- **per_user_allowed_item_ids** (*Optional[List[List[ItemIdType]]]*) – If not `None`, defines “a list of list of recommendable item IDs” and `len(allowed_item_ids)` must be equal to `score.shape[0]`. Defaults to `None`.
- **allowed_item_ids** – If not `None`, defines “a list of list of recommendable item IDs” and `len(allowed_item_ids)` must be equal to `len(item_ids)`. Defaults to `None`.
- **forbidden_item_ids** (*Optional[List[List[ItemIdType]]]*) – If not `None`, defines “a list of list of forbidden item IDs” and `len(allowed_item_ids)` must be equal to `len(item_ids)`. Defaults to `None`.
- **n_threads** (*Optional[int]*) –

Return type

List[List[Tuple[ItemIdType, float]]]

1.4.5 Dataset

<i>MovieLens1MDataManager</i>	Manages MovieLens 1M dataset.
<i>MovieLens100KDataManager</i>	Manages MovieLens 100K dataset.
<i>MovieLens20MDataManager</i>	
<i>NeuMFML1MDownloader</i>	Manages MovieLens 1M dataset split under 1-vs-100 negative evaluation protocol.

irspack.dataset.MovieLens1MDataManager

class irspack.dataset.**MovieLens1MDataManager**(zippath=None, force_download=False)

Bases: BaseMovieLenstDataLoader

Manages MovieLens 1M dataset.

Parameters

- **zippath** (*Optional[Union[Path, str]]*) – Where the zipped data is located. If *None*, assumes the path to be *~/ml-1m.zip*. If the designated path does not exist, you will be prompted for the permission to download the data. Defaults to *None*.
- **force_download** (*bool*) – If *True*, the class will not prompt for the permission and start downloading immediately.

__init__(zippath=None, force_download=False)

Specify the zip path for dataset. If that path does not exist, try downloading the relevant data from online resources.

Parameters

- **zippath** (*Optional[Union[Path, str]]*, *optional*) – *_description_*. Defaults to *None*.
- **force_download** (*bool*, *optional*) – *_description_*. Defaults to *False*.

Raises

RuntimeError – *_description_*

Methods

<code>__init__</code> ([zippath, force_download])	Specify the zip path for dataset.
<code>read_interaction</code> ()	Reads the entire user/movie/rating/timestamp interaction data.
<code>read_item_info</code> ()	
<code>read_user_info</code> ()	

Attributes

<code>DEFAULT_PATH</code>
<code>DOWNLOAD_URL</code>
<code>INTERACTION_PATH</code>
<code>ITEM_INFO_PATH</code>
<code>USER_INFO_PATH</code>

read_interaction()

Reads the entire user/movie/rating/timestamp interaction data.

Returns

The interaction *pd.DataFrame*, whose columns are [*“userId”, “movieId”, “rating”, “timestamp”*].

Return type

DataFrame

irspack.dataset.MovieLens100KDataManager

class irspack.dataset.**MovieLens100KDataManager**(zippath=None, force_download=False)

Bases: BaseMovieLenstDataLoader

Manages MovieLens 100K dataset.

Parameters

- **zippath** (*Optional[Union[Path, str]]*) – Where the zipped data is located. If *None*, assumes the path to be *~/ml-1m.zip*. If the designated path does not exist, you will be prompted for the permission to download the data. Defaults to *None*.
- **force_download** (*bool*) – If *True*, the class will not prompt for the permission and start downloading immediately.

__init__(zippath=None, force_download=False)

Specify the zip path for dataset. If that path does not exist, try downloading the relevant data from online resources.

Parameters

- **zippath** (*Optional[Union[Path, str]]*, *optional*) – *_description_*. Defaults to *None*.
- **force_download** (*bool*, *optional*) – *_description_*. Defaults to *False*.

Raises

RuntimeError – *_description_*

Methods

<code>__init__</code> (zippath, force_download)	Specify the zip path for dataset.
<code>read_interaction</code> ()	Reads the entire user/movie/rating/timestamp interaction data.
<code>read_item_info</code> ()	
<code>read_user_info</code> ()	

Attributes

DEFAULT_PATH

DOWNLOAD_URL

GENRE_PATH

INTERACTION_PATH

ITEM_INFO_PATH

USER_INFO_PATH

read_interaction()

Reads the entire user/movie/rating/timestamp interaction data.

Returns

The interaction *pd.DataFrame*, whose columns are [*“userId”, “movieId”, “rating”, “timestamp”*].

Return type

DataFrame

irspack.dataset.MovieLens20MDataManager

class irspack.dataset.**MovieLens20MDataManager**(*zippath=None, force_download=False*)

Bases: *BaseMovieLenstDataLoader*

Parameters

- **zippath** (*Optional[Union[Path, str]]*) –
- **force_download** (*bool*) –

__init__(*zippath=None, force_download=False*)

Specify the zip path for dataset. If that path does not exist, try downloading the relevant data from online resources.

Parameters

- **zippath** (*Optional[Union[Path, str]], optional*) – *_description_*. Defaults to *None*.
- **force_download** (*bool, optional*) – *_description_*. Defaults to *False*.

Raises

RuntimeError – *_description_*

Methods

<code>__init__([zippath, force_download])</code>	Specify the zip path for dataset.
<code>read_interaction()</code>	Reads the entire user/movie/rating/timestamp interaction data.

Attributes

<code>DEFAULT_PATH</code>
<code>DOWNLOAD_URL</code>
<code>INTERACTION_PATH</code>

`read_interaction()`

Reads the entire user/movie/rating/timestamp interaction data.

Returns

The interaction *pd.DataFrame*, whose columns are [*“userId”*, *“movieId”*, *“rating”*, *“timestamp”*].

Return type

DataFrame

`irspack.dataset.NeuMFML1MDownloader`

class `irspack.dataset.NeuMFML1MDownloader`(*zippath=None, force_download=False*)

Bases: `NeuMFDownloader`

Manages MovieLens 1M dataset split under 1-vs-100 negative evaluation protocol.

Parameters

- **zippath** (*Optional[Union[Path, str]]*) – Where the zipped data is located. If *None*, assumes the path to be *~/neumf-ml-1m.zip*. If the designated path does not exist, you will be prompted for the permission to download the data. Defaults to *None*.
- **force_download** (*bool*) – If *True*, the class will not prompt for the permission and start downloading immediately.

`__init__`(*zippath=None, force_download=False*)

Specify the zip path for dataset. If that path does not exist, try downloading the relevant data from online resources.

Parameters

- **zippath** (*Optional[Union[Path, str]]*, *optional*) – *_description_*. Defaults to *None*.
- **force_download** (*bool*, *optional*) – *_description_*. Defaults to *False*.

Raises

RuntimeError – *_description_*

Methods

<code>__init__([zippath, force_download])</code>	Specify the zip path for dataset.
<code>read_train_test()</code>	

Attributes

<code>DEFAULT_PATH</code>
<code>NEGATIVE_URL</code>
<code>TRAIN_URL</code>

INDICES AND TABLES

- `genindex`
- `search`

Symbols

- `__init__()` (*irspack.dataset.MovieLens100KDataManager* method), 120
 - `__init__()` (*irspack.dataset.MovieLens1MDataManager* method), 119
 - `__init__()` (*irspack.dataset.MovieLens20MDataManager* method), 121
 - `__init__()` (*irspack.dataset.NeuMFML1MDownloader* method), 122
 - `__init__()` (*irspack.evaluation.Evaluator* method), 19
 - `__init__()` (*irspack.evaluation.EvaluatorWithColdUser* method), 22
 - `__init__()` (*irspack.recommenders.AsymmetricCosineKNNRecommender* method), 61
 - `__init__()` (*irspack.recommenders.AsymmetricCosineUserKNNRecommender* method), 80
 - `__init__()` (*irspack.recommenders.BPRFMRecommender* method), 94
 - `__init__()` (*irspack.recommenders.BaseRecommender* method), 24
 - `__init__()` (*irspack.recommenders.CosineKNNRecommender* method), 57
 - `__init__()` (*irspack.recommenders.CosineUserKNNRecommender* method), 75
 - `__init__()` (*irspack.recommenders.DenseSLIMRecommender* method), 89
 - `__init__()` (*irspack.recommenders.IALSRecommender* method), 34
 - `__init__()` (*irspack.recommenders.JaccardKNNRecommender* method), 66
 - `__init__()` (*irspack.recommenders.MultVAERecommender* method), 101
 - `__init__()` (*irspack.recommenders.P3alphaRecommender* method), 42
 - `__init__()` (*irspack.recommenders.RP3betaRecommender* method), 47
 - `__init__()` (*irspack.recommenders.SLIMRecommender* method), 85
 - `__init__()` (*irspack.recommenders.TopPopRecommender* method), 28
 - `__init__()` (*irspack.recommenders.TruncatedSVDRecommender* method), 51
 - `__init__()` (*irspack.recommenders.TverskyIndexKNNRecommender* method), 71
 - `__init__()` (*irspack.split.UserTrainTestInteractionPair* method), 106
 - `__init__()` (*irspack.utils.IDMapper* method), 114
 - `__init__()` (*irspack.utils.ItemIDMapper* method), 111
- ## A
- AsymmetricCosineKNNRecommender* (class in *irspack.recommenders*), 61
 - AsymmetricCosineUserKNNRecommender* (class in *irspack.recommenders*), 80
- ## B
- BaseRecommender* (class in *irspack.recommenders*), 24
 - BPRFMRecommender* (class in *irspack.recommenders*), 94
- ## C
- compute_item_embedding()* (*irspack.recommenders.IALSRecommender* method), 37
 - compute_user_embedding()* (*irspack.recommenders.IALSRecommender* method), 37
 - concat()* (*irspack.split.UserTrainTestInteractionPair* method), 107
 - CosineKNNRecommender* (class in *irspack.recommenders*), 56
 - CosineUserKNNRecommender* (class in *irspack.recommenders*), 75
- ## D
- DenseSLIMRecommender* (class in *irspack.recommenders*), 89
- ## E
- Evaluator* (class in *irspack.evaluation*), 18
 - EvaluatorWithColdUser* (class in *irspack.evaluation*), 21

G

`get_item_embedding()`
 (*irspack.recommenders.BPRFMRecommender*
 method), 96
`get_item_embedding()`
 (*irspack.recommenders.IALSRecommender*
 method), 37
`get_item_embedding()`
 (*irspack.recommenders.TruncatedSVDRecommender*
 method), 52
`get_recommender_class()` (in module
 irspack.recommenders), 94
`get_score()` (*irspack.evaluation.Evaluator* *method*), 20
`get_score()` (*irspack.evaluation.EvaluatorWithColdUser*
 method), 23
`get_score()` (*irspack.recommenders.AsymmetricCosineKNNRecommender*
 method), 63
`get_score()` (*irspack.recommenders.AsymmetricCosineUserKNNRecommender*
 method), 81
`get_score()` (*irspack.recommenders.BaseRecommender*
 method), 25
`get_score()` (*irspack.recommenders.BPRFMRecommender*
 method), 97
`get_score()` (*irspack.recommenders.CosineKNNRecommender*
 method), 58
`get_score()` (*irspack.recommenders.CosineUserKNNRecommender*
 method), 76
`get_score()` (*irspack.recommenders.DenseSLIMRecommender*
 method), 90
`get_score()` (*irspack.recommenders.IALSRecommender*
 method), 37
`get_score()` (*irspack.recommenders.JaccardKNNRecommender*
 method), 67
`get_score()` (*irspack.recommenders.MultVAERecommender*
 method), 102
`get_score()` (*irspack.recommenders.P3alphaRecommender*
 method), 43
`get_score()` (*irspack.recommenders.RP3betaRecommender*
 method), 48
`get_score()` (*irspack.recommenders.SLIMRecommender*
 method), 86
`get_score()` (*irspack.recommenders.TopPopRecommender*
 method), 29
`get_score()` (*irspack.recommenders.TruncatedSVDRecommender*
 method), 53
`get_score()` (*irspack.recommenders.TverskyIndexKNNRecommender*
 method), 72
`get_score_block()` (*irspack.recommenders.AsymmetricCosineKNNRecommender*
 method), 63
`get_score_block()` (*irspack.recommenders.AsymmetricCosineUserKNNRecommender*
 method), 82
`get_score_block()` (*irspack.recommenders.BaseRecommender*
 method), 25
`get_score_block()` (*irspack.recommenders.BPRFMRecommender*
 method), 97
 method), 97
`get_score_block()` (*irspack.recommenders.CosineKNNRecommender*
 method), 58
`get_score_block()` (*irspack.recommenders.CosineUserKNNRecommender*
 method), 77
`get_score_block()` (*irspack.recommenders.DenseSLIMRecommender*
 method), 91
`get_score_block()` (*irspack.recommenders.IALSRecommender*
 method), 37
`get_score_block()` (*irspack.recommenders.JaccardKNNRecommender*
 method), 67
`get_score_block()` (*irspack.recommenders.MultVAERecommender*
 method), 103
`get_score_block()` (*irspack.recommenders.P3alphaRecommender*
 method), 43
`get_score_block()` (*irspack.recommenders.RP3betaRecommender*
 method), 48
`get_score_block()` (*irspack.recommenders.SLIMRecommender*
 method), 86
`get_score_block()` (*irspack.recommenders.TopPopRecommender*
 method), 29
`get_score_block()` (*irspack.recommenders.TruncatedSVDRecommender*
 method), 53
`get_score_block()` (*irspack.recommenders.TverskyIndexKNNRecommender*
 method), 72
`get_score_cold_user()`
 (*irspack.recommenders.AsymmetricCosineKNNRecommender*
 method), 63
`get_score_cold_user()`
 (*irspack.recommenders.AsymmetricCosineUserKNNRecommender*
 method), 82
`get_score_cold_user()`
 (*irspack.recommenders.BaseRecommender*
 method), 26
`get_score_cold_user()`
 (*irspack.recommenders.BPRFMRecommender*
 method), 97
`get_score_cold_user()`
 (*irspack.recommenders.CosineKNNRecommender*
 method), 58
`get_score_cold_user()`
 (*irspack.recommenders.CosineUserKNNRecommender*
 method), 77
`get_score_cold_user()`
 (*irspack.recommenders.DenseSLIMRecommender*
 method), 91
`get_score_cold_user()`
 (*irspack.recommenders.IALSRecommender*
 method), 38
`get_score_cold_user()`
 (*irspack.recommenders.JaccardKNNRecommender*
 method), 68
`get_score_cold_user()`
 (*irspack.recommenders.MultVAERecommender*

<i>method</i>), 103	<i>method</i>), 48
<code>get_score_cold_user()</code> (<i>irspack.recommenders.P3alphaRecommender</i> <i>method</i>), 44	<code>get_score_cold_user_remove_seen()</code> (<i>irspack.recommenders.SLIMRecommender</i> <i>method</i>), 87
<code>get_score_cold_user()</code> (<i>irspack.recommenders.RP3betaRecommender</i> <i>method</i>), 48	<code>get_score_cold_user_remove_seen()</code> (<i>irspack.recommenders.TopPopRecommender</i> <i>method</i>), 30
<code>get_score_cold_user()</code> (<i>irspack.recommenders.SLIMRecommender</i> <i>method</i>), 87	<code>get_score_cold_user_remove_seen()</code> (<i>irspack.recommenders.TruncatedSVDRecommender</i> <i>method</i>), 53
<code>get_score_cold_user()</code> (<i>irspack.recommenders.TopPopRecommender</i> <i>method</i>), 30	<code>get_score_cold_user_remove_seen()</code> (<i>irspack.recommenders.TverskyIndexKNNRecommender</i> <i>method</i>), 72
<code>get_score_cold_user()</code> (<i>irspack.recommenders.TruncatedSVDRecommender</i> <i>method</i>), 53	<code>get_score_from_user_embedding()</code> (<i>irspack.recommenders.BPRFMRecommender</i> <i>method</i>), 98
<code>get_score_cold_user()</code> (<i>irspack.recommenders.TverskyIndexKNNRecommender</i> <i>method</i>), 72	<code>get_score_from_user_embedding()</code> (<i>irspack.recommenders.IALSRecommender</i> <i>method</i>), 38
<code>get_score_cold_user_remove_seen()</code> (<i>irspack.recommenders.AsymmetricCosineKNNRecommender</i> <i>method</i>), 63	<code>get_score_from_user_embedding()</code> (<i>irspack.recommenders.TruncatedSVDRecommender</i> <i>method</i>), 53
<code>get_score_cold_user_remove_seen()</code> (<i>irspack.recommenders.AsymmetricCosineUserKNNRecommender</i> <i>method</i>), 82	<code>get_score_remove_seen()</code> (<i>irspack.recommenders.AsymmetricCosineKNNRecommender</i> <i>method</i>), 63
<code>get_score_cold_user_remove_seen()</code> (<i>irspack.recommenders.BaseRecommender</i> <i>method</i>), 26	<code>get_score_remove_seen()</code> (<i>irspack.recommenders.AsymmetricCosineUserKNNRecommender</i> <i>method</i>), 82
<code>get_score_cold_user_remove_seen()</code> (<i>irspack.recommenders.BPRFMRecommender</i> <i>method</i>), 97	<code>get_score_remove_seen()</code> (<i>irspack.recommenders.BaseRecommender</i> <i>method</i>), 26
<code>get_score_cold_user_remove_seen()</code> (<i>irspack.recommenders.CosineKNNRecommender</i> <i>method</i>), 58	<code>get_score_remove_seen()</code> (<i>irspack.recommenders.BPRFMRecommender</i> <i>method</i>), 98
<code>get_score_cold_user_remove_seen()</code> (<i>irspack.recommenders.CosineUserKNNRecommender</i> <i>method</i>), 77	<code>get_score_remove_seen()</code> (<i>irspack.recommenders.CosineKNNRecommender</i> <i>method</i>), 59
<code>get_score_cold_user_remove_seen()</code> (<i>irspack.recommenders.DenseSLIMRecommender</i> <i>method</i>), 91	<code>get_score_remove_seen()</code> (<i>irspack.recommenders.CosineUserKNNRecommender</i> <i>method</i>), 77
<code>get_score_cold_user_remove_seen()</code> (<i>irspack.recommenders.IALSRecommender</i> <i>method</i>), 38	<code>get_score_remove_seen()</code> (<i>irspack.recommenders.DenseSLIMRecommender</i> <i>method</i>), 91
<code>get_score_cold_user_remove_seen()</code> (<i>irspack.recommenders.JaccardKNNRecommender</i> <i>method</i>), 68	<code>get_score_remove_seen()</code> (<i>irspack.recommenders.IALSRecommender</i> <i>method</i>), 38
<code>get_score_cold_user_remove_seen()</code> (<i>irspack.recommenders.MultVAERecommender</i> <i>method</i>), 103	<code>get_score_remove_seen()</code> (<i>irspack.recommenders.JaccardKNNRecommender</i> <i>method</i>), 68
<code>get_score_cold_user_remove_seen()</code> (<i>irspack.recommenders.P3alphaRecommender</i> <i>method</i>), 44	<code>get_score_remove_seen()</code> (<i>irspack.recommenders.MultVAERecommender</i> <i>method</i>), 103
<code>get_score_cold_user_remove_seen()</code> (<i>irspack.recommenders.RP3betaRecommender</i>	<code>get_score_remove_seen()</code> (<i>irspack.recommenders.P3alphaRecommender</i>

method), 44

get_score_remove_seen()
(irspack.recommenders.RP3betaRecommender
method), 49

get_score_remove_seen()
(irspack.recommenders.SLIMRecommender
method), 87

get_score_remove_seen()
(irspack.recommenders.TopPopRecommender
method), 30

get_score_remove_seen()
(irspack.recommenders.TruncatedSVDRecommender
method), 54

get_score_remove_seen()
(irspack.recommenders.TverskyIndexKNNRecommender
method), 73

get_score_remove_seen_block()
(irspack.recommenders.AsymmetricCosineKNNRecommender
method), 64

get_score_remove_seen_block()
(irspack.recommenders.AsymmetricCosineUserKNNRecommender
method), 82

get_score_remove_seen_block()
(irspack.recommenders.BaseRecommender
method), 26

get_score_remove_seen_block()
(irspack.recommenders.BPRFMRecommender
method), 98

get_score_remove_seen_block()
(irspack.recommenders.CosineKNNRecommender
method), 59

get_score_remove_seen_block()
(irspack.recommenders.CosineUserKNNRecommender
method), 78

get_score_remove_seen_block()
(irspack.recommenders.DenseSLIMRecommender
method), 91

get_score_remove_seen_block()
(irspack.recommenders.IALSRecommender
method), 38

get_score_remove_seen_block()
(irspack.recommenders.JaccardKNNRecommender
method), 68

get_score_remove_seen_block()
(irspack.recommenders.MultVAERecommender
method), 103

get_score_remove_seen_block()
(irspack.recommenders.P3alphaRecommender
method), 44

get_score_remove_seen_block()
(irspack.recommenders.RP3betaRecommender
method), 49

get_score_remove_seen_block()
(irspack.recommenders.SLIMRecommender
method), 87

get_score_remove_seen_block()
(irspack.recommenders.TopPopRecommender
method), 30

get_score_remove_seen_block()
(irspack.recommenders.TruncatedSVDRecommender
method), 54

get_score_remove_seen_block()
(irspack.recommenders.TverskyIndexKNNRecommender
method), 73

get_scores() (irspack.evaluation.Evaluator method),
20

get_scores() (irspack.evaluation.EvaluatorWithColdUser
method), 23

get_target_score() (irspack.evaluation.Evaluator
method), 20

get_target_score() (irspack.evaluation.EvaluatorWithColdUser
method), 23

get_user_embedding()
(irspack.recommenders.BPRFMRecommender
method), 98

get_user_embedding()
(irspack.recommenders.IALSRecommender
method), 39

get_user_embedding()
(irspack.recommenders.TruncatedSVDRecommender
method), 54

H

holdout_specific_interactions() (in module
irspack.split), 109

I

IALSRecommender (class in irspack.recommenders), 32

IDMapper (class in irspack.utils), 114

ItemIDMapper (class in irspack.utils), 111

J

JaccardKNNRecommender (class in
irspack.recommenders), 66

L

learn() (irspack.recommenders.AsymmetricCosineKNNRecommender
method), 64

learn() (irspack.recommenders.AsymmetricCosineUserKNNRecommender
method), 83

learn() (irspack.recommenders.BaseRecommender
method), 27

learn() (irspack.recommenders.BPRFMRecommender
method), 98

learn() (irspack.recommenders.CosineKNNRecommender
method), 59

learn() (irspack.recommenders.CosineUserKNNRecommender
method), 78

`learn()` (*irspack.recommenders.DenseSLIMRecommender* method), 92
`learn()` (*irspack.recommenders.IALSRecommender* method), 39
`learn()` (*irspack.recommenders.JaccardKNNRecommender* method), 69
`learn()` (*irspack.recommenders.MultVAERecommender* method), 104
`learn()` (*irspack.recommenders.P3alphaRecommender* method), 45
`learn()` (*irspack.recommenders.RP3betaRecommender* method), 49
`learn()` (*irspack.recommenders.SLIMRecommender* method), 88
`learn()` (*irspack.recommenders.TopPopRecommender* method), 31
`learn()` (*irspack.recommenders.TruncatedSVDRecommender* method), 54
`learn()` (*irspack.recommenders.TverskyIndexKNNRecommender* method), 73
`learn_with_optimizer()` (*irspack.recommenders.AsymmetricCosineKNNRecommender* method), 64
`learn_with_optimizer()` (*irspack.recommenders.AsymmetricCosineUserKNNRecommender* method), 83
`learn_with_optimizer()` (*irspack.recommenders.BaseRecommender* method), 27
`learn_with_optimizer()` (*irspack.recommenders.BPRFMRecommender* method), 99
`learn_with_optimizer()` (*irspack.recommenders.CosineKNNRecommender* method), 59
`learn_with_optimizer()` (*irspack.recommenders.CosineUserKNNRecommender* method), 78
`learn_with_optimizer()` (*irspack.recommenders.DenseSLIMRecommender* method), 92
`learn_with_optimizer()` (*irspack.recommenders.IALSRecommender* method), 39
`learn_with_optimizer()` (*irspack.recommenders.JaccardKNNRecommender* method), 69
`learn_with_optimizer()` (*irspack.recommenders.MultVAERecommender* method), 104
`learn_with_optimizer()` (*irspack.recommenders.P3alphaRecommender* method), 45
`learn_with_optimizer()` (*irspack.recommenders.RP3betaRecommender* method), 49
`learn_with_optimizer()` (*irspack.recommenders.SLIMRecommender* method), 88
`learn_with_optimizer()` (*irspack.recommenders.TopPopRecommender* method), 31
`learn_with_optimizer()` (*irspack.recommenders.TruncatedSVDRecommender* method), 54
`learn_with_optimizer()` (*irspack.recommenders.TverskyIndexKNNRecommender* method), 73
`list_of_user_profile_to_matrix()` (*irspack.utils.IDMapper* method), 115
`list_of_user_profile_to_matrix()` (*irspack.utils.ItemIDMapper* method), 111

M

`MovieLens100KDataManager` (class in *irspack.dataset*), 100
`MovieLens1MDDataManager` (class in *irspack.dataset*), 119
`MovieLens20MDDataManager` (class in *irspack.dataset*), 121
`MultVAERecommender` (class in *irspack.recommenders*), 100

N

`n_items` (*irspack.split.UserTrainTestInteractionPair* attribute), 107
`n_users` (*irspack.split.UserTrainTestInteractionPair* attribute), 107
`NeuMFML1MDownloader` (class in *irspack.dataset*), 122

P

`P3alphaRecommender` (class in *irspack.recommenders*), 42

R

`read_interaction()` (*irspack.dataset.MovieLens100KDataManager* method), 121
`read_interaction()` (*irspack.dataset.MovieLens1MDDataManager* method), 119
`read_interaction()` (*irspack.dataset.MovieLens20MDDataManager* method), 122
`recommend_for_known_user_batch()` (*irspack.utils.IDMapper* method), 115
`recommend_for_known_user_id()` (*irspack.utils.IDMapper* method), 116
`recommend_for_new_user()` (*irspack.utils.IDMapper* method), 117

[recommend_for_new_user\(\)](#)
(irspack.utils.ItemIDMapper method), 112
[recommend_for_new_user_batch\(\)](#)
(irspack.utils.IDMapper method), 117
[recommend_for_new_user_batch\(\)](#)
(irspack.utils.ItemIDMapper method), 112
[rowwise_train_test_split\(\)](#) (in module
irspack.split), 108
[RP3betaRecommender](#) (class in *irspack.recommenders*),
46

S

[score_to_recommended_items_batch\(\)](#)
(irspack.utils.IDMapper method), 118
[score_to_recommended_items_batch\(\)](#)
(irspack.utils.ItemIDMapper method), 113
[SLIMRecommender](#) (class in *irspack.recommenders*), 84
[split_dataframe_partial_user_holdout\(\)](#) (in
module *irspack.split*), 108
[split_last_n_interaction_df\(\)](#) (in module
irspack.split), 110

T

[TopPopRecommender](#) (class in *irspack.recommenders*),
28
[TruncatedSVDRecommender](#) (class in
irspack.recommenders), 51
[tune\(\)](#) (*irspack.recommenders.AsymmetricCosineKNNRecommender*
class method), 64
[tune\(\)](#) (*irspack.recommenders.AsymmetricCosineUserKNNRecommender*
class method), 83
[tune\(\)](#) (*irspack.recommenders.BaseRecommender* class
method), 27
[tune\(\)](#) (*irspack.recommenders.BPRFMRecommender*
class method), 99
[tune\(\)](#) (*irspack.recommenders.CosineKNNRecommender*
class method), 60
[tune\(\)](#) (*irspack.recommenders.CosineUserKNNRecommender*
class method), 78
[tune\(\)](#) (*irspack.recommenders.DenseSLIMRecommender*
class method), 92
[tune\(\)](#) (*irspack.recommenders.IALSRecommender* class
method), 39
[tune\(\)](#) (*irspack.recommenders.JaccardKNNRecommender*
class method), 69
[tune\(\)](#) (*irspack.recommenders.MultVAERecommender*
class method), 104
[tune\(\)](#) (*irspack.recommenders.P3alphaRecommender*
class method), 45
[tune\(\)](#) (*irspack.recommenders.RP3betaRecommender*
class method), 50
[tune\(\)](#) (*irspack.recommenders.SLIMRecommender*
class method), 88

[tune\(\)](#) (*irspack.recommenders.TopPopRecommender*
class method), 31
[tune\(\)](#) (*irspack.recommenders.TruncatedSVDRecommender*
class method), 55
[tune\(\)](#) (*irspack.recommenders.TverskyIndexKNNRecommender*
class method), 74
[tune_doubling_dimension\(\)](#)
(irspack.recommenders.IALSRecommender
class method), 41
[TverskyIndexKNNRecommender](#) (class in
irspack.recommenders), 70

U

[U](#) (*irspack.recommenders.AsymmetricCosineUserKNNRecommender*
property), 81
[U](#) (*irspack.recommenders.CosineUserKNNRecommender*
property), 76
[UserTrainTestInteractionPair](#) (class in
irspack.split), 106

W

[W](#) (*irspack.recommenders.AsymmetricCosineKNNRecommender*
property), 62
[W](#) (*irspack.recommenders.CosineKNNRecommender*
property), 58
[W](#) (*irspack.recommenders.DenseSLIMRecommender*
property), 90
[W](#) (*irspack.recommenders.JaccardKNNRecommender*
property), 67
[W](#) (*irspack.recommenders.P3alphaRecommender* prop-
erty), 43
[W](#) (*irspack.recommenders.RP3betaRecommender* prop-
erty), 48
[W](#) (*irspack.recommenders.SLIMRecommender* property),
86
[W](#) (*irspack.recommenders.TverskyIndexKNNRecommender*
property), 72

X

[X_all](#) (*irspack.split.UserTrainTestInteractionPair* at-
tribute), 107
[X_test](#) (*irspack.split.UserTrainTestInteractionPair* at-
tribute), 107
[X_train](#) (*irspack.split.UserTrainTestInteractionPair* at-
tribute), 107
[X_train_all](#) (*irspack.recommenders.AsymmetricCosineKNNRecommender*
attribute), 62
[X_train_all](#) (*irspack.recommenders.AsymmetricCosineUserKNNRecommender*
attribute), 81
[X_train_all](#) (*irspack.recommenders.BaseRecommender*
attribute), 25
[X_train_all](#) (*irspack.recommenders.BPRFMRecommender*
attribute), 96

`X_train_all` (*irspack.recommenders.CosineKNNRecommender*
attribute), [58](#)

`X_train_all` (*irspack.recommenders.CosineUserKNNRecommender*
attribute), [76](#)

`X_train_all` (*irspack.recommenders.DenseSLIMRecommender*
attribute), [90](#)

`X_train_all` (*irspack.recommenders.IALSRecommender*
attribute), [37](#)

`X_train_all` (*irspack.recommenders.JaccardKNNRecommender*
attribute), [67](#)

`X_train_all` (*irspack.recommenders.MultVAERecommender*
attribute), [102](#)

`X_train_all` (*irspack.recommenders.P3alphaRecommender*
attribute), [43](#)

`X_train_all` (*irspack.recommenders.RP3betaRecommender*
attribute), [48](#)

`X_train_all` (*irspack.recommenders.SLIMRecommender*
attribute), [86](#)

`X_train_all` (*irspack.recommenders.TopPopRecommender*
attribute), [29](#)

`X_train_all` (*irspack.recommenders.TruncatedSVDRecommender*
attribute), [52](#)

`X_train_all` (*irspack.recommenders.TverskyIndexKNNRecommender*
attribute), [72](#)