LibRecommender

Release 1.0.0

massquantity

Feb 16, 2023

INTRO

LibRecommender is an easy-to-use recommender system focused on end-to-end recommendation process. It contains training(libreco) and serving(libserving) module for users to quickly train and deploy different kinds of recommendation models.

The main features are:

- Implements a number of popular recommendation algorithms such as FM, DIN, LightGCN etc. See full algorithm list.
- A hybrid recommender system, which allows users to use either collaborative-filtering or content-based features. New features can be added on the fly.
- Low memory usage, automatically convert categorical and multi-value categorical features to sparse representation.
- Support training for both explicit and implicit datasets, as well as negative sampling on implicit data.
- Provide end-to-end workflow, i.e. data handling / preprocessing -> model training -> evaluate -> save/load -> serving.
- Support cold-start prediction and recommendation.
- Provide unified and friendly API for all algorithms.
- Easy to retrain model with new users/items from new data.

CHAPTER

ONE

QUICK START

1. Pure collaborative-filtering example, which uses `LightGCN` model and includes process of train, evaluate, predict, recommend and cold-start:

Listing 1: From file examples/pure_example.py

```
# split whole data into three folds for training, evaluating and testing
train_data, eval_data, test_data = random_split(data, multi_ratios=[0.8, 0.1, 0.1])

train_data, data_info = DatasetPure.build_trainset(train_data)
eval_data = DatasetPure.build_evalset(eval_data)
test_data = DatasetPure.build_testset(test_data)

# sample negative items for each record
train_data.build_negative_samples(data_info)
eval_data.build_negative_samples(data_info)
test_data.build_negative_samples(data_info)
print(data_info)  # n_users: 5894, n_items: 3253, data sparsity: 0.4172 %

lightgcn = LightGCN(
    task="ranking",
    data_info=data_info,
    loss_type="bpr",
    embed_size=16,
    n_epochs=3,
    lr=1e-3,
    batch_size=2048,
    num_neg=1,
    device="cuda",
)
# monitor metrics on eval_data during training
lightgcn.fit(
    train_data,
    verbose=2,
    eval_data=eval_data,
    metrics=["loss", "roc_auc", "precision", "recall", "ndcg"],
)

# do final evaluation on test data
print(
    "evaluate_result: ",
```

(continues on next page)

(continued from previous page)

```
    evaluate(
        model=lightgcn,
        data=test_data,
        metrics=["loss", "roc_auc", "precision", "recall", "ndcg"],
    ),
)
# predict preference of user 2211 to item 110
print("prediction: ", lightgcn.predict(user=2211, item=110))
# recommend 7 items for user 2211
print("recommendation: ", lightgcn.recommend_user(user=2211, n_rec=7))

# cold-start prediction
print(
    "cold prediction: ",
    lightgcn.predict(user="ccc", item="not item", cold_start="average"),
)
# cold-start recommendation
print(
    "cold recommendation: ",
    lightgcn.recommend_user(user="are we good?", n_rec=7, cold_start="popular"),
)
```

2. With features example, which uses `YouTubeRanking` model and includes process of train, evaluate, predict, recommend and cold-start:

Listing 2: From file examples/feat_example.py

```
# split into train and test data based on time
train_data, test_data = split_by_ratio_chrono(data, test_size=0.2)

# specify complete columns information
sparse_col = ["sex", "occupation", "genre1", "genre2", "genre3"]
dense_col = ["age"]
user_col = ["sex", "age", "occupation"]
item_col = ["genre1", "genre2", "genre3"]

train_data, data_info = DatasetFeat.build_trainset(
    train_data, user_col, item_col, sparse_col, dense_col
)
test_data = DatasetFeat.build_testset(test_data)

# sample negative items for each record
train_data.build_negative_samples(data_info)
test_data.build_negative_samples(data_info)
print(data_info)  # n_users: 5962, n_items: 3226, data sparsity: 0.4185 %

ytb_ranking = YouTubeRanking(
    task="ranking",
    data_info=data_info,
    embed_size=16,
    n_epochs=3,
    lr=1e-4,
```

(continues on next page)

(continued from previous page)

```
        batch_size=512,
        use_bn=True,
        hidden_units=(128, 64, 32),
    )
    ytb_ranking.fit(
        train_data,
        verbose=2,
        shuffle=True,
        eval_data=test_data,
        metrics=["loss", "roc_auc", "precision", "recall", "map", "ndcg"],
    )

    # predict preference of user 2211 to item 110
    print("prediction: ", ytb_ranking.predict(user=2211, item=110))
    # recommend 7 items for user 2211
    print("recommendation: ", ytb_ranking.recommend_user(user=2211, n_rec=7))

    # cold-start prediction
    print(
        "cold prediction: ",
        ytb_ranking.predict(user="ccc", item="not item", cold_start="average"),
    )
    # cold-start recommendation
    print(
        "cold recommendation: ",
        ytb_ranking.recommend_user(user="are we good?", n_rec=7, cold_start="popular"),
    )
```

1.1 Installation

From pypi :

```
$ pip install LibRecommender
```

Build from source:

```
$ git clone https://github.com/massquantity/LibRecommender.git
$ cd LibRecommender
$ pip install .
```

Or if you want to modify some source code, e.g. implementing a new algorithm by inheriting from base classes in the library, you can also use editable installs, which allows you to modify the source code and have the changes take effect without having to rebuild and reinstall (The `-vv` flag is used for outputting the build process)

```
$ git clone https://github.com/massquantity/LibRecommender.git
$ cd LibRecommender
$ pip install -e . -vv
```

1.1.1 Dependencies

Hint: LibRecommender contains two modules: libreco for `training` and libserving for `serving`. If one only wants to train a model, dependencies for *libserving* are not needed.

Caution: Since version `1.0.0`, the following dependencies will **NOT** be installed along with LibRecommender to avoid messing up your local dependencies.

Please make sure dependencies in your machine meet the version requirements. Or one can manually run the requirements file:

```
$ pip install -r requirements.txt
```

to install all the `libreco` dependencies.

Dependencies for libreco:

- Python >= 3.6
- TensorFlow >= 1.15
- PyTorch >= 1.10
- Numpy >= 1.19.5
- Pandas >= 1.0.0
- Scipy >= 1.2.1
- scikit-learn >= 0.20.0
- gensim >= 4.0.0
- tqdm
- nmslib (optional, see *Embedding*)
- DGL (optional, see *Implementation Details*)
- Cython >= 0.29.0 (optional, for building source)

Note: If you are using Python 3.6, you also need to install dataclasses, which was first introduced in Python 3.7.

Known issue: Sometimes one may encounter errors like `ValueError: numpy.ndarray size changed, may indicate binary incompatibility. Expected 88 from C header, got 80 from PyObject`. In this case try upgrading numpy, and version 1.22.0 or higher is probably a safe option.

Dependencies for libserving:

- Python >= 3.7
- sanic >= 22.3
- requests
- aiohttp
- pydantic
- ujson
- redis
- redis-py >= 4.2.0
- faiss >= 1.5.2
- TensorFlow Serving == 2.8.2

1.2 Tutorial

This tutorial will walk you through the comprehensive process of training a model in LibRecommender, i.e. **data processing -> feature engineering -> training -> evaluate -> save/load -> retrain**. We will use Wide & Deep as the example algorithm.

First make sure LibRecommender is installed.

```
$ pip install LibRecommender
```

Serving

For how to deploy a trained model in LibRecommender, see *Serving Guide*.

TensorFlow1 issue

If you encounter errors like `Variables already exist, disallowed...` in this tutorial, just call `tf.compat.v1.reset_default_graph()` first. It's one of the inconvenience from TensorFlow1.

1.2.1 Load Data

In this tutorial we will use the MovieLens 1M dataset. The following code will load the data into `pandas.DataFrame` format. If the data does not exist locally, it will be downloaded at first.

```
import random
import warnings
import zipfile
from pathlib import Path
from urllib.request import urlretrieve

import pandas as pd
```

(continues on next page)

(continued from previous page)

```
import tensorflow as tf
import tqdm
warnings.filterwarnings("ignore")
```

```
def split_genre(line):
    genres = line.split("|")
    if len(genres) == 3:
        return genres[0], genres[1], genres[2]
    elif len(genres) == 2:
        return genres[0], genres[1], "missing"
    elif len(genres) == 1:
        return genres[0], "missing", "missing"
    else:
        return "missing", "missing", "missing"
```

```
def load_ml_1m():
    download_path = "http://files.grouplens.org/datasets/movielens/ml-1m.zip"
    original_file = "ml-1m.zip"
    cur_path = Path(".").absolute()
    if not Path.exists(Path(original_file)):
        print("Data does not exist, start downloading...")
        with tqdm.tqdm(unit='B', unit_scale=True) as p:
            def report(chunk, chunksize, total):
                p.total = total
                p.update(chunksize)
            urlretrieve(download_path, original_file, reporthook=report)
        print("Download successful!")
    # extract zip file
    with zipfile.ZipFile(original_file, 'r') as f:
        f.extractall(cur_path)

    # read and merge data into same table
    ratings = pd.read_csv(
        cur_path / "ml-1m" / "ratings.dat",
        sep="::",
        usecols=[0, 1, 2, 3],
        names=["user", "item", "rating", "time"],
    )
    users = pd.read_csv(
        cur_path / "ml-1m" / "users.dat",
        sep="::",
        usecols=[0, 1, 2, 3],
        names=["user", "sex", "age", "occupation"],
    )
    items = pd.read_csv(
        cur_path / "ml-1m" / "movies.dat",
        sep="::",
        usecols=[0, 2],
        names=["item", "genre"],
        encoding="iso-8859-1",
    )
```

(continues on next page)

(continued from previous page)

```
    items["genre1"], items["genre2"], items["genre3"] = zip(*items["genre"].apply(split_
↪genre))
    items.drop("genre", axis=1, inplace=True)
    data = ratings.merge(users, on="user").merge(items, on="item")
    data.rename(columns={"rating": "label"}, inplace=True)
    return data
```

```
>>> data = load_ml_1m()
>>> data.shape
```

```
data shape: (1000209, 10)
```

```
>>> data.iloc[random.choices(range(len(data)), k=10)]  # randomly select 10 rows
```

Now we have about 1 million data. In order to perform evaluation after training, we need to split the data into train, eval and test data first. In this tutorial we will simply use *random_split()*. For other ways of splitting data, see *Data Processing*.

Note: For now, We will only use **first half data** for training. Later we will use the rest data to retrain the model.

1.2.2 Process Data & Features

```
>>> from libreco.data import random_split

# split data into three folds for training, evaluating and testing
>>> first_half_data = data[: (len(data) // 2)]
>>> train_data, eval_data, test_data = random_split(first_half_data, multi_ratios=[0.8,␣
↪0.1, 0.1], seed=42)
```

```
>>> print("first half data shape:", first_half_data.shape)
```

```
first half data shape: (500104, 10)
```

The data contains some categorical features such as "sex" and "genre", as well as a numerical feature "age". In LibRecommender we use `sparse_col` to represent categorical features and `dense_col` to represent numerical features. So one should specify the column information and then use `DatasetFeat.build_*` functions to process the data.

```
>>> from libreco.data import DatasetFeat

>>> sparse_col = ["sex", "occupation", "genre1", "genre2", "genre3"]
>>> dense_col = ["age"]
>>> user_col = ["sex", "age", "occupation"]
>>> item_col = ["genre1", "genre2", "genre3"]

>>> train_data, data_info = DatasetFeat.build_trainset(train_data, user_col, item_col,␣
↪sparse_col, dense_col)
>>> eval_data = DatasetFeat.build_evalset(eval_data)
>>> test_data = DatasetFeat.build_testset(test_data)
```

`user_col` means features belong to user, and `item_col` means features belong to item. Note that the column numbers should match, i.e. `len(sparse_col) + len(dense_col) == len(user_col) + len(item_col)`.

```
>>> print(data_info)
```

```
n_users: 6040, n_items: 3576, data density: 1.8523 %
```

In this example we treat all the samples in data as positive samples, and perform negative sampling. This is a standard procedure for "implicit data".

```
# sample negative items for each record
>>> train_data.build_negative_samples(data_info)
>>> eval_data.build_negative_samples(data_info)
>>> test_data.build_negative_samples(data_info)
```

1.2.3 Training the Model

Now with all the data and features prepared, we can start training the model!

Since as its name suggests, the `Wide & Deep` algorithm has wide and deep parts, and they use different optimizers. So we should specify the learning rate separately by using a dict: `{"wide": 0.01, "deep": 3e-4}`. For other model hyper-parameters, see API reference of *WideDeep*.

```
from libreco.algorithms import WideDeep
```

```
model = WideDeep(
    task="ranking",
    data_info=data_info,
    embed_size=16,
    n_epochs=2,
    loss_type="cross_entropy",
    lr={"wide": 0.01, "deep": 3e-4},
    batch_size=2048,
    use_bn=True,
    hidden_units=(128, 64, 32),
)

model.fit(
    train_data,
    verbose=2,
    shuffle=True,
    eval_data=eval_data,
    metrics=["loss", "roc_auc", "precision", "recall", "ndcg"],
)
```

```
Epoch 1 elapsed: 3.053s
    train_loss: 0.6778
    eval log_loss: 0.5676
    eval roc_auc: 0.8005
    eval precision@10: 0.0277
    eval recall@10: 0.0409
    eval ndcg@10: 0.1119
```

(continues on next page)

(continued from previous page)

```
Epoch 2 elapsed: 3.008s
    train_loss: 0.4994
    eval log_loss: 0.4928
    eval roc_auc: 0.8373
    eval precision@10: 0.0321
    eval recall@10: 0.0506
    eval ndcg@10: 0.1384
```

We've trained the model for 2 epochs and evaluated the performance on the eval data during training. Next we can evaluate on the *independent* test data.

```
>>> from libreco.evaluation import evaluate
>>> evaluate(model=model, data=test_data, metrics=["loss", "roc_auc", "precision",
↪"recall", "ndcg"])
```

```
{'loss': 0.49392982253743395,
 'roc_auc': 0.8364561294428758,
 'precision': 0.03056640625,
 'recall': 0.05029253291880213,
 'ndcg': 0.12794099009836263}
```

1.2.4 Make Recommendation

The recommend part is pretty straightforward. You can make recommendation for one user or a batch of users.

```
>>> model.recommend_user(user=1, n_rec=3)
```

```
{1: array([ 260, 2858, 1210])}
```

```
>>> model.recommend_user(user=[1, 2, 3], n_rec=3)
```

```
{1: array([ 260, 2858, 1210]),
 2: array([527, 356, 480]),
 3: array([ 589, 2571, 1240])}
```

1.2.5 Save, Load and Inference

When saving the model, we should also save the `data_info` for feature information.

```
>>> data_info.save("model_path", model_name="wide_deep")
>>> model.save("model_path", model_name="wide_deep")
```

Then we can load the model and make recommendation again.

```
>>> tf.compat.v1.reset_default_graph()  # need to reset graph in TensorFlow1
```

```
>>> from libreco.data import DataInfo

>>> loaded_data_info = DataInfo.load("model_path", model_name="wide_deep")
>>> loaded_model = WideDeep.load("model_path", model_name="wide_deep", data_info=loaded_
↪data_info)
>>> loaded_model.recommend_user(user=1, n_rec=3)
```

1.2.6 Retrain the Model with New Data

Remember that we split the original `MovieLens 1M` data into *two parts* in the first place? We will treat the **second half** of the data as our new data and retrain the saved model with it. In real-world recommender systems, data may be generated every day, so it is inefficient to train the model from scratch every time we get some new data.

```
>>> second_half_data = data[(len(data) // 2) :]
>>> train_data, eval_data = random_split(second_half_data, multi_ratios=[0.8, 0.2])
```

```
>>> print("second half data shape:", second_half_data.shape)
```

```
second half data shape: (500105, 10)
```

The data processing is similar, except that we should use *merge_trainset()* and *merge_evalset()* in *DatasetFeat*.

The purpose of these functions is combining information from old data with that from new data, especially for the possible new users/items from new data. For more details, see *Model Retrain*.

```
>>> train_data = DatasetFeat.merge_trainset(train_data, loaded_data_info, merge_
↪behavior=True)  # use loaded_data_info
>>> eval_data = DatasetFeat.merge_evalset(eval_data, loaded_data_info)

>>> train_data.build_negative_samples(loaded_data_info, seed=2022)  # use loaded_data_
↪info
>>> eval_data.build_negative_samples(loaded_data_info, seed=2222)
```

Then we construct a new model, and call *rebuild_model()* method to assign the old trained variables into the new model.

```
>>> tf.compat.v1.reset_default_graph()  # need to reset graph in TensorFlow1
```

```
new_model = WideDeep(
    task="ranking",
    data_info=loaded_data_info,  # pass loaded_data_info
    embed_size=16,
    n_epochs=2,
    loss_type="cross_entropy",
    lr={"wide": 0.01, "deep": 3e-4},
    batch_size=2048,
    use_bn=True,
    hidden_units=(128, 64, 32),
)
```

(continues on next page)

(continued from previous page)

```
new_model.rebuild_model(path="model_path", model_name="wide_deep", full_assign=True)
```

Finally, the training and recommendation parts are the same as before.

```
new_model.fit(
    train_data,
    verbose=2,
    shuffle=True,
    eval_data=eval_data,
    metrics=["loss", "roc_auc", "precision", "recall", "ndcg"],
)
```

```
Epoch 1 elapsed: 2.955s
    train_loss: 0.4604
    eval log_loss: 0.4497
    eval roc_auc: 0.8678
    eval precision@10: 0.1015
    eval recall@10: 0.0715
    eval ndcg@10: 0.3106

Epoch 2 elapsed: 2.657s
    train_loss: 0.4332
    eval log_loss: 0.4371
    eval roc_auc: 0.8760
    eval precision@10: 0.1043
    eval recall@10: 0.0740
    eval ndcg@10: 0.3189
```

```
>>> new_model.recommend_user(user=1, n_rec=3)
```

```
{1: array([2858, 1259, 3175])}
```

```
>>> new_model.recommend_user(user=[1, 2, 3], n_rec=3)
```

```
{1: array([2858, 1259, 3175]),
 2: array([1222, 1240,  858]),
 3: array([2858, 1580,  589])}
```

This completes our tutorial!

Where to go from here

For more examples, see the examples/ folder on GitHub.

For more usages, please head to user_guide/index.

For serving a trained model, please head to *Python Serving Guide*.

1.3 Data Processing

1.3.1 Data Format

Just normal data format, each line represents a sample. One thing is important, the model assumes that `user`, `item`, and `label` column index are 0, 1, and 2, respectively. You may wish to change the column order if that's not the case.

If you have only one data, you can split the data in following ways:

- `random_split`. Split the data randomly.
- `split_by_ratio`. For each user, assign a certain ratio of items to test_data.
- `split_by_num`. For each user, assign a certain number of items to test_data.
- `split_by_ratio_chrono`. For each user, assign certain ratio of items to test_data, where items are sorted by time first. In this case, data should contain a `time` column.
- `split_by_num_chrono`. For each user, assign certain number of items to test_data, where items are sorted by time first. In this case, data should contain a `time` column.

See also:

split_data_example.py

Caution: Some caveats about the data:

1. Your data should not contain any missing value. Otherwise, it may lead to unexpected behavior.
2. If your data size is small (less than 10,000 rows), the four `split_by_*` function may not be suitable. Since the number of interacted items for each user may be only one or two, which makes it difficult to split the whole data. In this case `random_split` is more suitable.
3. Some data may contain duplicate samples, e.g., a user may have clicked an item multiple times. In this case, the training and possible negative sampling will be done multiple times for the same sample. If you don't want this, consider using functions such as drop_duplicates in Pandas before training.

1.3.2 Task

There are generally two kinds of tasks in LibRecommender , i.e. `rating` and `ranking` task. The `rating` task deals with explicit data such as `MovieLens` or `Netflix` dataset, whereas the `ranking` task deals with implicit data such as Last.FM dataset. The main difference on usage between these two tasks are:

1. The `task` parameter must be specified when building a model.
2. Obviously the metrics used for evaluating should be different. For `rating` task, the available metrics are [`rmse`, `mae`, `r2`] , and for `ranking` task the available metrics are [`loss`, `balanced_accuracy`, `roc_auc`, `pr_auc`, `precision`, `recall`, `map`, `ndcg`] .

For example, using the `SVD` model with `rating` task:

```
>>> model = SVD(task="rating", ...)
>>> model.fit(..., metrics=["rmse", "mae", "r2"])
```

The implicit data typically may only contain positive feedback, i.e. only has samples that labeled as 1. In this case, negative sampling is needed to effectively train a model.

By the way, some models such as BPR , YouTubeRetrieval, YouTubeRanking, Item2Vec, DeepWalk, LightGCN etc. , can only be used for ranking tasks since they are specially designed for that. Errors might be raised if one use them for rating task.

1.3.3 Negative Sampling

For implicit data with only positive labels, negative sampling is typically used in model training. There are some special cases, such as UserCF, ItemCF, BPR, YouTubeRetrieval, RNN4Rec with bpr loss, where these models do not need to do negative sampling during training.

However, when evaluating these models using some metrics such as cross_entropy loss, roc_auc, pr_auc, negative labels are indeed needed.

For PyTorch-based models, **only eval or test data needs negative sampling**. These models includes NGCF, LightGCN, GraphSage, GraphSageDGL, PinSage, PinSageDGL , see torch_ranking_example.py.

For other models, performing negative sampling on all the train, eval and test data is recommended as long as **your data is implicit and only contains positive labels**.

```
>>> train_data.build_negative_samples(data_info, item_gen_mode="random", num_neg=1,␣
↪seed=2020)
>>> test_data.build_negative_samples(data_info, item_gen_mode="random", num_neg=1,␣
↪seed=2222)
```

TODO

In the future, we plan to remove this explicit negative sampling part before training. This requires encapsulating the sampling process into the batch training, so that users won't undertake the ambiguity above. Some other sampling methods apart from random will also be added.

1.4 Feature Engineering

1.4.1 Sparse and Dense features

Sparse features are typically categorical features such as sex, location, year, etc. These features are projected into low dimension vectors by using an embedding layer, and this is by far the most common way of handling these kinds of features.

Dense features are typically numerical features such as age, price, length, etc. Unfortunately, there is no common way of handling these features, so in LibRecommender we mainly use the method described in the AutoInt paper.

Specifically, every dense feature are also projected into low dimension vectors through an embedding layer, then the vectors are multiplied by the dense feature value itself. In this way, the authors of the paper argued that sparse and dense features can have interactions in models such as FM, DeepFM and of course, AutoInt.

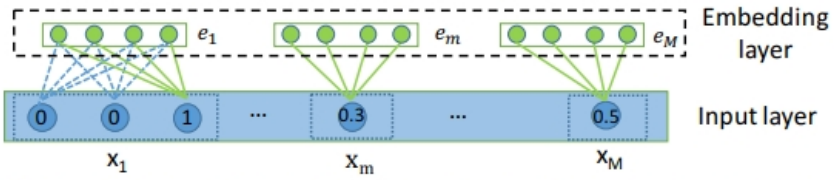

Figure 2: Illustration of input and embedding layer, where both categorical and numerical fields are represented by low-dimensional dense vectors.

So to be clear, for one dense feature, all samples of this feature will be projected into a same embedding vector. This is different from a sparse feature, where all samples of it will have different embedding vectors based on its concrete category.

Apart from `sparse` and `dense` features, `user` and `item` features should also be provided. Since in order to make predictions and recommendations, the model needs to know whether a feature belongs to user or item. So, in short, these parameters are [`sparse_col`, `dense_col`, `user_col`, `item_col`].

1.4.2 `Multi_Sparse` features

Often times categorical features can be multi-valued. For example, a movie may have multiple genres, as shown in the `genre` feature in the `MovieLens-1m` dataset:

```
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
```

Usually we can handle this kind of feature by using multi-hot encoding, so in LibRecommender they are called `multi_sparse` features. After some transformation, the data can become like this (just for illustration purpose):

item_id	movie_name	genre1	genre2	genre3
1	Toy Story (1995)	Animation	Children's	Comedy
2	Jumanji (1995)	Adventure	Children's	Fantasy
3	Grumpier Old Men (1995)	Comedy	Romance	missing
4	Waiting to Exhale (1995)	Comedy	Drama	missing
5	Father of the Bride Part II (1995)	Comedy	missing	missing

In this case, a `multi_sparse_col` can be used:

```
multi_sparse_col = [["genre1", "genre2", "genre3"]]
```

Note it's a list of list, because there are possibly many multi_sparse features, for instance, `[[a1, a2, a3], [b1, b2]]` .

When you specify a feature as `multi_sparse` feature like this, each sub-feature, i.e. `genre1`, `genre2`, `genre3` in the table above, will share the same embedding of the original feature `genre`. Whether the embedding sharing would improve the model performance is data-dependent. But one thing is certain, it will reduce the total number of model parameters.

LibRecommender provides multiple ways of dealing with `multi_sparse` features, i.e. `normal`, `sum` , `mean` and `sqrtn`. `normal` means treating each sub-feature's embedding separately, and in most cases they will be concatenated at last. `sum` and `mean` means computing the sum or mean of each sub-feature's embedding, in this case they are combined as one feature. `sqrtn` means the result of `sum` divided by the square root of sub-feature number, e.g. sqrt(3) in `genre` feature. I'm not sure about this, but I think this `sqrtn` idea originally came from SVD++, and it was also used in *Scaled Dot-Product Attention* part of Transformer. Generally the four methods described here have similar functionality as in tf.nn.embedding_lookup_sparse, but we didn't use it directly in our implementation since it has no `normal` choice.

So in general you should choose a strategy in parameter `multi_sparse_combiner` when building models with `multi_sparse` features:

```
>>> model = DeepFM(..., multi_sparse_combiner="sqrtn")  # other options: normal, sum,␣
↪mean
```

Note the `genre` feature above has different number of sub-features among all the samples. Some movie only has one genre, whereas others may have three. So the value "missing" is used to pad them into same length. However, when using `sum`, `mean` or `sqrtn` to combine these sub-features, the padding value should be excluded. Thus, you can pass the `pad_val` parameter when building the data, and the model will do all the work. Otherwise, the padding value will be included in the transformed features.

```
>>> train_data, data_info = DatasetFeat.build_trainset(multi_sparse_col=[["genre1",
↪"genre2", "genre3"]], pad_val=["missing"])
```

Although here we use "missing" as the padding value, this is not always appropriate. It is fine with `str` type, but with numerical features, a value with corresponding type should be used. e.g. 0 or -999.99.

Also be aware that the `pad_val` parameter is a list and should have the same length as the number of `multi_sparse` features, and the reason for this is obvious. So all in all an example script is enough to illustrate the usage of `multi_sparse` features, see multi_sparse_example.py.

LibRecommender also provides a convenient function (`split_multi_value`) to transform the original `multi_sparse` features to the divided sub-features illustrated above.

Listing 3: From file examples/multi_sparse_processing_example.py

```
    multi_sparse_col, multi_user_col, multi_item_col = split_multi_value(
        data,
        multi_value_col,
        sep="|",
        max_len=[3],
        pad_val="missing",
        user_col=user_col,
        item_col=item_col,
    )
```

1.4.3 Changing Feature

In real-world scenarios, users' features are very likely to change every time we make recommendations for them. For example, a user's location may change many times a day, and we may need to take this into account. This feature issue can actually be combined with the cold-start issue. For example, a user has appeared in training data, but his/her location doesn't exist in training data's `location` feature.

How do we handle these changing feature problems? Fortunately, LibRecommender can deal with them elegantly.

If you want to predict or recommend with specific features, the usage is pretty straightforward. For prediction, just pass the `feats` argument, which only accepts `dict` or `pands.Series` type:

```
>>> model.predict(user=1, item=110, feats={"sex": "F", "occupation": 2, "age": 23})
```

There is no need to specify a feature belongs to user or item, because this information has already been stored in model's *DataInfo* object. Note if you misspelled some feature names, e.g. "sex" -> "sax", the model will simply ignore this feature. If you pass a feature category that doesn't appear in training data, e.g. "sex" -> "bisexual", then it will be ignored too.

If you want to predict on a whole dataset with features, you can use the `predict_data_with_feats` function. By setting `batch_size` to `None`, the model will treat all the data as one batch, which may cause memory issues:

```
>>> from libreco.prediction import predict_data_with_feats
>>> predict_data_with_feats(model, data=dataset, batch_size=1024, cold_start="average")
```

To make recommendation for one user, we can pass the user features to `user_feats` argument. It actually doesn't make much sense to change the item features when making recommendation for only one user, but we provide an `item_data` argument anyway, which can change the item features. The type of `item_data` must be `pandas.DataFrame` . We assume one may want to change the features of multiple items, since it nearly makes no difference to the recommendation result if only one item's features have been changed.

```
>>> model.recommend_user(user=1, n_rec=7, cold_start="popular",
                         user_feats=pd.Series({"sex": "F", "occupation": 2, "age": 23}),
                         item_data=item_features)
```

Note the three functions described above doesn't change the unique user/item features inside the *DataInfo* object. So the next time you call `model.predict(user=1, item=110)` , it will still use the features stored in `DataInfo`. However, if you do want to change the features in `DataInfo`, then you can use `assign_user_features` and `assign_item_features` :

```
>>> data_info.assign_user_features(user_data=data)
>>> data_info.assign_item_features(item_data=data)
```

The passed `data` argument is a `pandas.DataFrame` that contains the user/item information. Be careful with this assign operation if you are not sure if the features in `data` are useful.

See also:

changing_feature_example.py

1.5 Model & Train

1.5.1 `Pure` and `Feat` model

LibRecommender is a hybrid recommender system, which means you can choose whether to use features other than user behaviors or not. For models only use user behaviors, we classify them as `pure` models. This category includes `UserCF`, `ItemCF`, `SVD`, `SVD++`, `ALS`, `NCF`, `BPR`, `RNN4Rec`, `Item2Vec`, `Caser`, `WaveNet`, `DeepWalk`, `NGCF`, `LightGCN`.

Then for models that can include other features (e.g., age, sex, name etc.), we call them `feat` models. This category includes `WideDeep`, `FM`, `DeepFM`, `YouTubeRetrieval`, `YouTubeRanking`, `AutoInt`, `DIN`, `GraphSage`, `PinSage`.

The main difference on usage between these two kinds of models are:

1. `pure` models should use *DatasetPure* to process data, and `feat` models should use *DatasetFeat* to process data.

2. When using `feat` models, four parameters should be provided, i.e. [`sparse_col`, `dense_col`, `user_col`, `item_col`], as otherwise the model will have no idea how to deal with all kinds of features.

The `fit()` method is the sole method for training a model in LibRecommender. You can find some typical usages in these examples:

See also:

- pure_rating_example.py
- pure_ranking_example.py
- feat_rating_example.py
- feat_ranking_example.py

In fact, there exists two other kinds of model categories in LibRecommender, and we call them `sequence` and `graph` models. You can find them in the algorithm list.

Sequence models leverage information of user behavior sequence, whereas Graph models leverage information from graph. As you can see, these models overlap with `pure` and `feat` models. But no need to worry, the APIs remain the same, and you can just refer to the examples above.

1.5.2 Loss

LibRecommender provides some options on loss type for `ranking` *task*. The default loss type for `ranking` is cross entropy loss. Since version `0.10.0`, focal loss was added into the library. First introduced in Lin et al., 2018, focal loss down-weights well-classified examples and focuses on hard examples to get better training performance, and here is the implementation.

In order to choose which loss to use, simply set the `loss_type` parameter:

```
>>> model = Caser(task="ranking", loss_type="cross_entropy", ...)
>>> model = Caser(task="ranking", loss_type="focal", ...)
```

There are some special cases:

- Some algorithms are hard to assign explicit loss type, including `UserCF`, `ItemCF`, `ALS`, `Item2Vec`, `DeepWalk`, so they don't have `loss_type` parameter.
- As its name suggests, `BPR` can only use `bpr` loss.
- The `YouTubeRetrieval` algorithm is also different, its `loss_type` is either `sampled_softmax` or `nce`.
- Finally, with `RNN4Rec` algorithm, one can choose three `loss_type`, i.e. `cross_entropy`, `focal`, `bpr`.

We are aware that these loss restrictions are hard to remember at once, so this leaves room for further improvements:)

1.6 Evaluation & Save/Load

1.6.1 Evaluate During Training

The standard procedure in LibRecommender is evaluating during training. However, for some complex models doing full evaluation on eval data can be very time-consuming, so you can specify some evaluation parameters to speed this up.

The default value of `eval_batch_size` is 8192, and you can use a higher value if you have enough machine or GPU memory. On the contrary, if you encounter memory error during evaluation, try reducing `eval_batch_size`.

The `eval_user_num` parameter controls how many users to use in evaluation. By default, it is `None`, which uses all the users in eval data. You can use a smaller value if the evaluation is slow, and this will sample `eval_user_num` users randomly from eval data.

```
model.fit(
    train_data,
    verbose=2,
    shuffle=True,
    eval_data=eval_data,
    metrics=metrics,
    k=10,  # parameter of metrics, e.g. recall at k, ndcg at k
    eval_batch_size=8192,
    eval_user_num=100,
)
```

1.6.2 Evaluate After Training

After the training, one can use the *evaluate()* function to evaluate on test data directly.

By default, it also won't update features stored in *DataInfo*, but you can choose `update_features=True` to achieve that. Also note if your evaluation data(typically in `pandas.DataFrame` format) **is implicit and only contains positive label**, then negative sampling is needed by passing `neg_sample=True`:

Listing 4: From file examples/save_load_example.py

```
    data = pd.read_csv("sample_data/sample_movielens_merged.csv", sep=",", header=0)
    train, test = split_by_ratio_chrono(data, test_size=0.2)
    eval_result = evaluate(
        model=model,
        data=test,
        eval_batch_size=8192,
        k=10,
        metrics=["roc_auc", "precision"],
        sample_user_num=2048,
        neg_sample=True,
        update_features=False,
        seed=2222,
    )
```

1.6.3 Save/Load Model

In general, we may want to save/load a model for two reasons:

1. Save the model, then load it to make some predictions and recommendations. This is called inference.
2. Save the model, then load it to retrain the model when we get some new data.

The `save/load` API mainly deal with the first one, and the retraining problem is quite different, which will be covered in the *Model Retrain*. When making predictions and recommendations, it may be unnecessary to save all the model variables. So one can pass `inference_only=True` to only save the essential model part.

After loading the model, one can also evaluate the model directly, see save_load_example.py for typical usages.

1.7 Recommendation

By default, the recommendation result returned by `model.recommend_user()` method will filter out items that a user has previously consumed.

However, if you use a very large `n_rec` and number of consumed items for this user plus `n_rec` exceeds number of total items, i.e. `len(user_consumed) + n_rec > n_items`, the consumed items will not be filtered out since there are not enough items to recommend. If you don't want to filter out consumed items, set `filter_consumed=False`.

LibRecommender also supports random recommendation by setting `random_rec=True` (By default it is False). Of course, it's not completely random, but random sampling based on each item's prediction scores. It's basically a trade-off between accuracy and diversity.

Finally, batch recommendation is also supported by simply passing a list to the `user` parameter. The returned result will be a dict, with users as keys and `numpy.array` as values.

```
>>> model.recommend_user(user=[1, 2, 3], n_rec=3, filter_consumed=True, random_rec=False)
# returns {1: array([2529, 1196, 2916]), 2: array([ 541,  750, 1299]), 3: array([3183,␣
↪2722, 2672])}
```

1.7.1 Cold Start

It is very common to encounter new users or items that doesn't exist in training data, which is hard to make recommendations for them. This is the notorious "cold-start" problem in recommender system.

There are two strategies in LibRecommender to handle the cold-start problem: `popular` and `average`. The `popular` strategy simply returns the most popular items in training data.

The `average` strategy means using the average of all the user/item embeddings as the representation of the cold-start user/item. Once we have the embedding, we can make predictions and recommendations. This strategy indicates that a cold-start user/item's behavior is treated as the "average" behavior of all the known users/items.

Likewise, the new category of one feature are also handled as an average embedding of the known categories of this feature. See pure_example.py, feat_example.py for cold-start usage.

See also:

Embedding

1.8 Embedding

According to the algorithm list, there are some algorithms that can generate user and item embeddings after training. So LibRecommender provides public APIs to get them:

```
>>> model = RNN4Rec(task="ranking", ...)
>>> model.fit(train_data, ...)
>>> model.get_user_embedding(user=1)  # get user embedding for user 1
>>> model.get_item_embedding(item=2)  # get item embedding for item 2
```

One can also search for similar users/items based on embeddings. By default, we use nmslib to do approximate similarity searching since it's generally fast, but some people may find it difficult to build and install the library, especially on Windows platform or Python >= 3.10. So one can fall back to numpy similarity calculation if nmslib is not available.

```
>>> model = RNN4Rec(task="ranking", ...)
>>> model.fit(train_data, ...)
>>> model.init_knn(approximate=True, sim_type="cosine")
>>> model.search_knn_users(user=1, k=3)
>>> model.search_knn_items(item=2, k=3)
```

Before searching, one should call *init_knn()* to initialize the index. Set `approximate=True` if you can use nmslib, otherwise set `approximate=False`. The `sim_type` parameter should either be `cosine` or `inner-product`.

See also:

knn_embedding_example.py

1.9 Model Retrain

1.9.1 The Problem

When we get some new data, we definitely want to retrain the old model with these new data, but this is actually not easy for some deep learning models.

The reason lies in the new users/items that may appear in the new data. In deep learning models, embedding variables are commonly used, and their shapes are preassigned and fixed during and after training. The same issue also goes with new features. If we load the old model and want to train it with new users/items/features, these embedding shape must be expanded, which is not allowed in TensorFlow and PyTorch (Well, at least to my knowledge).

One workaround is to combine the new data with the old data, then retrain the model with all the data. But it would be a waste of time and resources to retrain on the whole data every time we get some new data.

1.9.2 Bigger Problem?

So how can we retrain a model if we can't change the shape of the variables in TensorFlow and PyTorch? Well, if we can't alter it, we create a new one, then explicitly assign the old one to the new one. Specifically, in TensorFlow, we build a new graph with variables with new shape, then assign the old values to the correct indices of new variables. For the new indices, i.e. the new user/item part, they are initialized randomly as usual.

The problem with this solution is that we can not use TensorFlow's default method such as tf.train.Saver or tf.saved_model, as well as PyTorch's default method such as load_state_dict , since these can only load to the exact same model with same shapes.

Things become even more desperate if we also want to save and restore the optimizers' variables, e.g. the first and second moment in Adam optimizer. Since these variables are used as states in optimizers, failing to keep them means losing the previous training state.

1.9.3 Solution in LibRecommender

So our solution is extracting all the variables to `numpy.ndarray` format, then save them using the save method in numpy. After that the variables are loaded from numpy, we then build the new graph and update the new variables with old ones.

So it's crucial to set `manual=True, inference_only=False` when you save the model, which means leveraging the numpy way. If you set `manual=False`, the model may use the `tf.train.Saver` or torch.save to save the model, which is OK if you are certain that there will be no new user/item in new data.

Listing 5: From file examples/model_retrain_example.py

```
deepfm.save(
    path="model_path", model_name="deepfm_model", manual=True, inference_only=False
)
```

Before retraining the model, the new data should also be transformed. Since the old `data_info` already exists, we need to merge new information with the old one, especially those new users/items/features from new data. This achieved by calling *`merge_trainset()`*, *`merge_evalset()`*, *`merge_testset()`* functions.

During recommendation, we usually want to filter some items that a user has previously consumed, which are also stored in *`DataInfo`* object. So if you want to combine the user-consumed information in old data with that in new data, you can pass `merge_behavior=True`:

```
>>> train_data = DatasetFeat.merge_trainset(train, data_info, merge_behavior=True)
```

Finally, loading the old variables and assigning them to the new model requires only one function *`rebuild_model()`*:

```
>>> model.rebuild_model(path="model_path", model_name="deepfm_model", full_assign=True)
```

See also:

model_retrain_example.py

1.10 Python Serving Guide

1.10.1 Introduction

This guide mainly describes how to serve a trained model using the libserving module in LibRecommender. Prior to LibRecommender version `0.10.0`, Flask was used to construct the serving web server. However, to take full advantage of the asynchronous feature and modern `async/await` syntax in Python, we've decided to switch to Sanic framework. Unlike flask, a sanic server can run in production directly, and the typical command is like this:

```
$ sanic server:app --host=127.0.0.1 --port=8000 --dev --access-logs -v --workers 1  #␣
↪develop mode
$ sanic server:app --no-access-logs --workers 10  # production mode
```

Refer to Running Sanic for more details.

Rust

Beyond Python, one can also use Rust to serve a model. See *Rust Serving Guide*.

From model serving's perspective, currently there are three kinds of models in LibRecommender:

- knn-based model
- embed-based model
- tensorflow-based model.

As for models trained with PyTorch, they all belong to embed-based model.

The following is the main serving workflow:

1. Serialize the trained model to disk.
2. Load model and save to redis.
3. Run the sanic server.
4. Make http request to the server and gain recommendation.

Here we choose NOT to save the trained model directly to redis, since: 1) Even you save the model to redis in the first place, you'll end up with saving to disk anyway :) 2) We try to keep the requirements of the main `libreco` module as minimal as possible.

So during serving, one should start redis server first:

```
$ redis-server
```

Error: Note that sometimes using redis in model serving can be error-prone:

For example, you served a `DeepFM` model at first, and later on you decided to use another `pure` model, say `NCF`. Since `DeepFM` is a `feat` model, some feature information may have been saved into redis. If you forget to remove these feature information before using `NCF`, the server may mistakenly load it and eventually causing an error.

Attention: In this guide we assume the following codes are all executed in `LibRecommender/libserving` folder, so one needs to clone the repository first:

```
$ git clone https://github.com/massquantity/LibRecommender.git
$ cd LibRecommender/libserving
```

1.10.2 Note about Dependencies

The serving related dependencies are listed in main README.

- redis-py introduced async support since 4.2.0.
- According to the official doc, faiss can't be installed from pip directly. But someone has built wheel packages, refer to faiss-wheels. So now the pip option is available for faiss.
- We use TensorFlow Serving to serve tensorflow-based models, and typically it is installed through Docker. However, The latest TensorFlow Serving might not work in some cases, and we have encountered similar error described in this issue on TensorFlow Serving 2.9 and 2.10. So for now the workable version is 2.8.2, and one should pull docker image like this:

```
$ sudo docker pull tensorflow/serving:2.8.2
```

1.10.3 Saving Format

In `libserving`, the primary data serialization format is JSON.

Aside from JSON, models built upon TensorFlow are saved using its own tf.saved_model API. The `SavedModel` format provides a language-neutral format to save machine-learning models.

1.10.4 KNN-based Model

KNN-based models refer to the classic `UserCF` and `ItemCF` algorithms, which leverage a similarity matrix to find similar users/items for recommendation. Due to the large number of users/items, it is often impractical to store the whole similarity matrix, so here we may only save the most similar k neighbors for each user/item.

Below is an example usage which saves 10 neighbors per item using ItermCF. One should also specify model-saving `path` :

```
>>> from libreco.algorithms import ItemCF
>>> from libreco.data import DatasetPure
>>> from libserving.serialization import knn2redis, save_knn

>>> train_data, data_info = DatasetPure.build_trainset(...)
>>> model = ItemCF(...)
>>> model.fit(...)  # train model
>>> path = "knn_model"  # specify model saving directory
>>> save_knn(path, model, k=10)  # save model in json format
>>> knn2redis(path, host="localhost", port=6379, db=0)  # load json from path and save␣
↪model to redis
```

```
$ sanic sanic_serving.knn_deploy:app --dev --access-logs -v --workers 1  # run sanic␣
↪server

# make requests
$ python request.py --user 1 --n_rec 10 --algo knn
$ curl -d '{"user": 1, "n_rec": 10}' -X POST http://127.0.0.1:8000/knn/recommend
# {'Recommend result for user 1': ['480', '589', '2571', '260', '2028', '1198', '1387', '1214', '1291
↪', '1197']}
```

1.10.5 Embed-based Model

Embed-based models perform similarity searching on embeddings to make recommendation, so we only need to save a bunch of embeddings. This kind of model includes `SVD`, `SVD++`, `ALS`, `BPR`, `YouTubeRetrieval`, `Item2Vec`, `DeepWalk`, `RNN4Rec`, `Caser`, `WaveNet`, `NGCF`, `LightGCN`, `GraphSage`, `PinSage`.

In practice, to speed up serving, some ANN(Approximate Nearest Neighbors) libraries are often used to find similar embeddings. Here in `libserving`, we use faiss to do such thing.

Below is an example usage which uses `ALS`. One should also specify model-saving `path`:

```
>>> from libreco.algorithms import ALS
>>> from libreco.data import DatasetPure
>>> from libserving.serialization import embed2redis, save_embed

>>> train_data, data_info = DatasetPure.build_trainset(...)
```

(continues on next page)

(continued from previous page)

```
>>> model = ALS(...)
>>> model.fit(...)  # train model
>>> path = "embed_model"  # specify model saving directory
>>> save_embed(path, model)  # save model in json format
>>> embed2redis(path, host="localhost", port=6379, db=0)  # load json from path and save␣
↪model to redis
```

The following code will train faiss index on model's item embeddings and save to disk as file name `faiss_index.bin`. The saved index will be loaded in sanic server.

```
>>> from libserving.serialization import save_faiss_index
>>> save_faiss_index(path, model)
```

```
$ sanic sanic_serving.embed_deploy:app --dev --access-logs -v --workers 1  # run sanic␣
↪server

# make requests
$ python request.py --user 1 --n_rec 10 --algo embed
$ curl -d '{"user": 1, "n_rec": 10}' -X POST http://127.0.0.1:8000/embed/recommend
# {'Recommend result for user 1': ['593', '1270', '318', '2858', '1196', '2571', '1617', '260', '1200
↪', '457']}
```

1.10.6 TensorFlow-based Model

As stated above, tensorflow-based model will typically be saved in `SavedModel` format. These model mainly contains neural networks, including `NCF`, `WideDeep`, `FM`, `DeepFM`, `YouTubeRanking` , `AutoInt` , `DIN`.

We assume TensorFlow Serving has already been installed through Docker. After successfully starting the docker container, we can post request to the serving model inside the sanic server and get the recommendation.

Below is an example usage which uses `DIN`. One should also specify model-saving `path`:

```
>>> from libreco.algorithms import DIN
>>> from libreco.data import DatasetFeat
>>> from libserving.serialization import save_tf, tf2redis

>>> train_data, data_info = DatasetFeat.build_trainset(...)
>>> model = DIN(...)
>>> model.fit(...)  # train model
>>> path = "tf_model"  # specify model saving directory
>>> save_tf(path, model, version=1)  # save model in json format
>>> tf2redis(path, host="localhost", port=6379, db=0)  # load json from path and save␣
↪model to redis
```

The directory of `SavedModel` format for a `DIN` model has the following structure and note that 1 is the version number:

```
din/
    1/
        variables/
            variables.data-?????-of-?????
            variables.index
        saved_model.pb
```

We can inspect the saved DIN model by using SavedModel CLI described in official doc. By default, it is bundled with TensorFlow. The following command will output:

```
$ saved_model_cli show --dir tf_model/din/1 --all
```

```
MetaGraphDef with tag-set: 'serve' contains the following SignatureDefs:

signature_def['predict']:
  The given SavedModel SignatureDef contains the following input(s):
    inputs['dense_values'] tensor_info:
        dtype: DT_FLOAT
        shape: (-1, 1)
        name: Placeholder_6:0
    inputs['item_indices'] tensor_info:
        dtype: DT_INT32
        shape: (-1)
        name: Placeholder_1:0
    inputs['sparse_indices'] tensor_info:
        dtype: DT_INT32
        shape: (-1, 5)
        name: Placeholder_5:0
    inputs['user_indices'] tensor_info:
        dtype: DT_INT32
        shape: (-1)
        name: Placeholder:0
    inputs['user_interacted_len'] tensor_info:
        dtype: DT_FLOAT
        shape: (-1)
        name: Placeholder_3:0
    inputs['user_interacted_seq'] tensor_info:
        dtype: DT_INT32
        shape: (-1, 10)
        name: Placeholder_2:0
  The given SavedModel SignatureDef contains the following output(s):
    outputs['logits'] tensor_info:
        dtype: DT_FLOAT
        shape: (-1)
        name: Reshape_4:0
  Method name is: tensorflow/serving/predict
```

The above result shows this DIN model needs 6 inputs, i.e. user_indices, item_indices, sparse_indices, dense_values, user_interacted_seq, user_interacted_len. But this only applies to DIN and other models may have different inputs.

- For NCF model, only user_indices and item_indices are needed since it's a collaborative-filtering algorithm.
- For WideDeep, FM, DeepFM, AutoInt, since they don't use behavior sequence information, 4 inputs are needed: user_indices, item_indices, sparse_indices, dense_values.
- Finally, YouTubeRanking has same inputs as DIN. They both use behavior sequence information.

However, these are just general cases. Suppose your data doesn't have any sparse feature, then it would be a mistake to feed the sparse_indices input, so these matters should be taken into account. This is exactly where a library fits in, and LibRecommender can dynamically handle these different feature situations. So as a library user, all you need to do is specifying the correct model path.

Using `SavedModel CLI`, we can even pass some inputs to the model and get outputs (note the inputs num should match the model requirement):

```
$ inputs="user_indices=np.int32([2,3]);item_indices=np.int32([2,3]);sparse_indices=np.
↪int32([[1,1,1,1,1],[1,1,1,1,1]]);dense_values=np.float32([[1],[2]]);user_interacted_
↪len=np.float32([2,3]);user_interacted_seq=np.int32([[1,2,3,4,5,6,7,8,9,10],[1,2,3,4,5,
↪6,7,8,9,10]])"

$ saved_model_cli run --dir tf_model/din/1 --tag_set serve --signature_def predict --
↪input_exprs $inputs
```

```
Result for output key logits:
[-0.51893234 -0.569685  ]
```

Now let's start TensorFlow Serving service through docker. Note that the `MODEL_NAME` should be lowercase of the model class name. For instance, `DIN` -> `din`, `YouTubeRanking` -> `youtuberanking`, `WideDeep` -> `widedeep`.

```
$ MODEL_NAME=din
$ MODEL_PATH=tf_model
$ sudo docker run --rm -t -p 8501:8501 --mount type=bind,source=$(pwd),target=$(pwd) -e␣
↪MODEL_BASE_PATH=$(pwd)/${MODEL_PATH} -e MODEL_NAME=${MODEL_NAME} tensorflow/serving:2.
↪8.2
```

Get model status from TensorFlow Serving service using RESTful API:

```
$ curl http://localhost:8501/v1/models/din
```

```
{
 "model_version_status": [
  {
   "version": "1",
   "state": "AVAILABLE",
   "status": {
    "error_code": "OK",
    "error_message": ""
   }
  }
 ]
}
```

Make predictions for two samples from TensorFlow Serving service:

```
$ curl -d '{"signature_name": "predict", "inputs": {"user_indices": [1, 2], "item_indices
↪": [3208, 2], "sparse_indices": [[1, 19, 32, 59, 71], [1, 19, 32, 59, 71]], "dense_
↪values": [22.0, 56.0], "user_interacted_seq": [[996, 1764, 2083, 520, 2759, 334, 304,␣
```

(continues on next page)

(continued from previous page)

```
↪1110, 2013, 1415],[996, 1764, 2083, 520, 2759, 334, 304, 1110, 2013, 1415]], "user_
↪interacted_len": [3, 10]}}' -X POST http://localhost:8501/v1/models/din:predict
```

```
{
    "outputs": [
        -0.65978992,
        -0.759211063
    ]
}
```

Now we can start the corresponding sanic server. According to the official doc, the input tensors can use either row format or column format. In tf_deploy.py we use column format since it's more compact.

```
$ sanic sanic_serving.tf_deploy:app --dev --access-logs -v --workers 1  # run sanic␣
↪server

# make requests
$ python request.py --user 1 --n_rec 10 --algo tf
$ curl -d '{"user": 1, "n_rec": 10}' -X POST http://127.0.0.1:8000/tf/recommend
# {'Recommend result for user 1': ['1196', '480', '260', '2028', '1198', '1214', '780', '1387', '1291
↪', '1197']}
```

1.11 Rust Serving Guide

1.11.1 Introduction

This guide mainly describes how to serve a trained model in LibRecommender with Rust. A Rust web server is typically much faster than its Python counterpart. In the libserving module, we use Actix, one of the fastest web frameworks in the world. The overall serving procedure in this guide resembles *Python Serving Guide*, so you'll need a Redis too. If you are already familiar with Rust, then follow the steps below. Otherwise, you can also use *Docker Compose*.

Users need to provide three environment variables before starting the server:

- `PORT` assigned to the server. In Actix 8080 is commonly used.
- `MODE_TYPE` specifies model category used in server. According to Python Serving Guide, the value should be one of `knn`, `embed`, `tf`.
- `WORKERS` specifies number of workers used in server.

1.11.2 KNN-based Model

```
$ cd LibRecommender/libserving
```

```
>>> from libreco.algorithms import ItemCF
>>> from libreco.data import DatasetPure
>>> from libserving.serialization import knn2redis, save_knn

>>> train_data, data_info = DatasetPure.build_trainset(...)
```

(continues on next page)

(continued from previous page)

```
>>> model = ItemCF(...)
>>> model.fit(...)  # train model
>>> path = "knn_model"  # specify model saving directory
>>> save_knn(path, model, k=10)  # save model in json format
>>> knn2redis(path, host="localhost", port=6379, db=0)  # load json from path and save␣
↪model to redis
```

```
$ cd actix_serving
$ export PORT=8080 MODEL_TYPE=knn WORKERS=8  # assign environment variables
```

Use either develop or production mode:

```
$ cargo run --package actix_serving --bin actix_serving  # develop mode
```

```
$ cargo build --release  # production mode
$ ./target/release/actix_serving
```

Note that the "user" parameter in request should be string and the route is knn/recommend :

```
$ curl -d '{"user": "1", "n_rec": 10}' -H "Content-Type: application/json" -X POST http:/
↪/0.0.0.0:8080/knn/recommend
# {"rec_list":["1806","1687","3799","807","1303","1860","3847","3616","1696","1859"]}
```

1.11.3 Embed-based model

Similar to Python Serving Guide, we use faiss to find similar embeddings. However, faiss can't be installed directly as in Python, so we use Rust bindings to Faiss. According to the instructions, one should fork the c_api_head branch and build faiss from source manually before including the crate. We should warn you first, this process can be pretty frustrating, and if you get stuck, you can view the Dockerfile-rs file to get some hints:). Or you can just save all these troubles and use *Docker Compose*.

After successfully installing Rust faiss, i.e. copy the c_api/libfaiss_c.so and faiss/libfaiss.so to LD_LIBRARY_PATH, the rest is straightforward:

```
$ cd LibRecommender/libserving
```

```
>>> from libreco.algorithms import ALS
>>> from libreco.data import DatasetPure
>>> from libserving.serialization import embed2redis, save_embed

>>> train_data, data_info = DatasetPure.build_trainset(...)
>>> model = ALS(...)
>>> model.fit(...)  # train model
>>> path = "embed_model"  # specify model saving directory
>>> save_embed(path, model)  # save model in json format
>>> embed2redis(path, host="localhost", port=6379, db=0)  # load json from path and save␣
↪model to redis
```

```
>>> from libserving.serialization import save_faiss_index
>>> save_faiss_index(path, model)  # save faiss index to disk, note this uses python␣
↪faiss, not rust faiss
```

```
$ cd actix_serving
$ export PORT=8080 MODEL_TYPE=embed WORKERS=8  # assign environment variables
```

Use either develop or production mode:

```
$ cargo run  # develop mode, this uses rust faiss
```

```
$ cargo build --release  # production mode
$ ./target/release/actix_serving
```

Note that the "user" parameter in request should be string and the route is embed/recommend :

```
$ curl -d '{"user": "1", "n_rec": 10}' -H "Content-Type: application/json" -X POST http:/
↪/0.0.0.0:8080/embed/recommend
# {"rec_list":["858","260","2355","1287","527","2371","1220","377","1968","3362"]}
```

1.11.4 TensorFlow-based model

```
$ cd LibRecommender/libserving
```

```
>>> from libreco.algorithms import DIN
>>> from libreco.data import DatasetFeat
>>> from libserving.serialization import save_tf, tf2redis

>>> train_data, data_info = DatasetFeat.build_trainset(...)
>>> model = DIN(...)
>>> model.fit(...)  # train model
>>> path = "tf_model"  # specify model saving directory
>>> save_tf(path, model, version=1)  # save model in json format
>>> tf2redis(path, host="localhost", port=6379, db=0)  # load json from path and save␣
↪model to redis
```

```
$ MODEL_NAME=din # variables for tensorflow serving
$ MODEL_PATH=tf_model
$ sudo docker run --rm -t -p 8501:8501 --mount type=bind,source=$(pwd),target=$(pwd) -e␣
↪MODEL_BASE_PATH=$(pwd)/${MODEL_PATH} -e MODEL_NAME=${MODEL_NAME} tensorflow/serving:2.
↪8.2  # start tensorflow serving
```

```
$ cd actix_serving
$ export PORT=8080 MODEL_TYPE=tf WORKERS=8 # assign environment variables
```

Use either develop or production mode:

```
$ cargo run  # develop mode, this uses rust faiss
```

```
$ cargo build --release  # production mode
$ ./target/release/actix_serving
```

1.11.5 Serving with Docker Compose

In docker-compose-rs.yml file, change the corresponding `MODEL_TYPE` environment variable. You may also need to change the path of volumes if model is stored in other place. Also, Redis has already been included, so you don't need a Redis server locally. But one should start the docker compose *before* saving data to Redis. For example in embed-based models:

```
$ cd LibRecommender/libserving
```

```
>>> from libreco.algorithms import ALS
>>> from libreco.data import DatasetPure
>>> from libserving.serialization import embed2redis, save_embed

>>> train_data, data_info = DatasetPure.build_trainset(...)
>>> model = ALS(...)
>>> model.fit(...)  # train model
>>> path = "embed_model"  # specify model saving directory
>>> save_embed(path, model)  # save model in json format

>>> from libserving.serialization import save_faiss_index
>>> save_faiss_index(path, model)  # save faiss index to disk
```

```
$ sudo docker compose -f docker-compose-rs.yml up  # start docker compose, which will␣
↪load faiss index
```

```
>>> embed2redis(path, host="0.0.0.0", port=6379, db=0)  # now load json from path and␣
↪save model to redis
```

```
$ curl -d '{"user": "1", "n_rec": 10}' -H "Content-Type: application/json" -X POST http:/
↪/0.0.0.0:8080/embed/recommend
# {"rec_list":["2628","1552","260","969","2193","733","1573","1917","1037","10"]}
```

For TensorFlow-based models, TensorFlow Serving config is in docker-compose-tf-serving.yml file, so we need to start them both, and don't forget to change the `MODEL_BASE_PATH` and `MODEL_NAME` environment variables in the file:

```
$ sudo docker compose -f docker-compose-rs.yml -f docker-compose-tf-serving.yml up
```

```
$ curl -d '{"user": "1", "n_rec": 10}' -H "Content-Type: application/json" -X POST http:/
↪/0.0.0.0:8080/tf/recommend
# {"rec_list":["858","260","2355","1287","527","2371","1220","377","1968","3362"]}
```

1.12 Internal

1.12.1 Implementation Details

In this section we describe some implementation details for algorithms in LibRecommender. In general, we try to follow the same settings used in the reference papers when implementing these algorithms. But in some cases we find it useful to change or extend these settings for better performance or speed, or it is necessary to adjust them in order to fit in with the whole process in LibRecommender. As we will explain in detail below.

UserCF / ItemCF

The traditional implementation of UserCF / ItemCF is pre-allocating a user-user or item-item similarity matrix, then computing similarities between all users/items and fill in the matrix. However, this can be problematic for big data, because allocating a full user-user or item-item matrix may use a lot of memory. For example, for only about 100 thousand items, a (100,000, 100,000) matrix of numpy float64 will consume approximately 70 GB memory. So in LibRecommender we mainly use scipy sparse matrices to store similarity matrix and save memory.

Furthermore, computing all the user-user or item-item similarities requires iterating all data in for loops, which can be extremely slow in pure Python, so we use Cython with multi-threading to bypass the Python GIL. Also for the same purpose, we apply the inverted index technic when computing similarities for better speed. The implementation is in _similarities.pyx, and users can choose whether to use forward index or inverted index by setting the `mode` parameter:

```
>>> model = UserCF(..., mode="invert") # or "forward", note that "forward" mode can be␣
↪much slower than "invert" mode
```

The concrete similarity metrics are cosine, pearson and jaccard. `pearson` is more suitable for rating task, and `jaccard` is more suitable for ranking task, whereas `cosine` is suitable for both. Users can choose them by setting `sim_type` parameter:

```
>>> model = ItemCF(..., sim_type="cosine")
```

FM / DeepFM

The FM and FM part of DeepFM is actually an implementation of NFM, which generalizes FM. The main difference is that in FM the final dimension of interaction layer is added element-wisely, whereas in NFM it is fed into DNN and gets final prediction. We found that NFM had a slightly better performance than FM.

YouTubeRetrieval / YouTubeRanking

YouTubeRetrieval corresponds to the *candidate generation* stage of the paper, whereas YouTubeRanking corresponds to the *ranking* stage. For YouTubeRetrieval, The paper stated that negative sampling was used to alleviate extreme multi-class problem. So in our implementation tf.nn.sampled_softmax_loss and tf.nn.nce_loss are used, and you can choose one of them by specifying the `loss_type` parameter. However, there are two caveats when using these sampling techniques in TensorFlow:

By default, the `sampled_values` in tf.nn.sampled_softmax_loss and tf.nn.nce_loss uses tf.random.log_uniform_candidate_sampler, which samples candidates based on log-uniform distribution. This basically means items with higher popularity will be more likely sampled. This is actually not very surprising because sampled-softmax and nce are originally came from NLP area, and in NLP this setting is common. However, this may not be suitable in recommender system scenario, especially in large-scale retrieval problem. So in LibRecommender you can set the `sampler` parameter to `uniform` to make the model use tf.random.uniform_candidate_sampler, which

samples items from uniform distribution. The default value in `sampler` is indeed `uniform`, and if you change it to other value, the default log-uniform in TensorFlow will be used.

Another caveat worth mentioning is the `num_sampled` parameter in tf.nn.sampled_softmax_loss and tf.nn.nce_loss. The doc states that this parameter means "The number of classes to randomly sample **per batch**". Well, this is counter-intuitive, because in LibRecommender the `num_neg` parameter means "number of negative samples per positive sample". If you set `num_sampled` to 1, the model will sample only 1 negative sample for a batch data. So the parameter controls this setting in LibRecommender is `num_sampled_per_batch`, and the default value is `batch_size`, which means every positive sample in a batch will get 1 negative sample. But of course you can change to the value you want.

```
model = YouTubeRetrieval(
    task,
    data_info,
    loss_type="sampled_softmax",  # or "nce"
    num_sampled_per_batch=None,  # "None" will use batch_size, or can be set to 1, 1000,␣
↪9999 ...
    sampler="uniform",
)
```

RNN4Rec / GRU4Rec

The paper stated that they used GRU for modeling session-based data. But of course we can use LSTM too by setting the `rnn_type` parameter.

```
>>> model= RNN4Rec(task, data_info, rnn_type="lstm") # or "gru"
```

WaveNet

At first glance it looks weird to have WaveNet in LibRecommender, since it's a model used for generating raw audio. But if you look at the paper closely, the way they model audio waveforms using CNN can also be applied to user behavior sequence. So we can generate user embedding based on this technique.

NGCF / LightGCN

The NGCF and LightGCN paper used BPR (*Bayesian Personalized Ranking*) loss, but in LibRecommender one can also choose other losses by setting the `loss_type` parameter.

```
ngcf = NGCF(
    "ranking",
    data_info,
    loss_type="cross_entropy",  # or "focal", "bpr", "max_margin"
)
lightgcn = LightGCN(
    "ranking",
    data_info,
    loss_type="bpr",
)
```

PinSage

In LibRecommender, there are two versions of PinSage implementation: PyTorch and DGL version. Since some users may find it difficult to install DGL on Windows platform (see issue), we provide an additional PyTorch version. In general the DGL version is much faster, but the PyTorch version can have more control over sampling process.

The paper used max-margin loss on item-item inner product score. We extend this setting in our implementation. In recommender system scenario this is called "i2i", and the other form is "u2i", which is also commonly used and combines user features and item features to compute scores. The parameter for controlling this is `paradigm`.

Max-margin loss belongs to pairwise loss, but we can also use other losses. In LibRecommender you can use `cross_entropy`, `focal`, `bpr`, `max_margin` by setting the `loss_type` parameter.

Another important extension in LibRecommender is that users can choose which features to use freely, instead of using domain-specific features described in the paper. So you can use PinSage just like other `feat` models:

```
>>> sparse_col = ["sex", "occupation", "genre1", "genre2", "genre3"]
>>> dense_col = ["age"]
>>> user_col = ["sex", "age", "occupation"]
>>> item_col = ["genre1", "genre2", "genre3"]
>>> train_data, data_info = DatasetFeat.build_trainset(train, user_col, item_col, sparse_
↪col, dense_col)

>>> from libreco.algorithms import PinSage, PinSageDGL
>>> model = PinSage(  # PyTorch version
        task,
        data_info,
        loss_type="cross_entropy",  # or "focal", "bpr", "max_margin"
        paradigm="u2i",  # or "i2i"
    )
>>> model = PinSageDGL(  # DGL version
        task,
        data_info,
        loss_type="max_margin",
        paradigm="i2i",
    )
>>> model.fit(train_data)
```

GraphSage

GraphSage was not originally designed for recommender system problem, but we have adapted it to fit in with LibRecommender. Just like PinSage, GraphSage also has PyTorch and DGL version. The main difference between them is that the PyTorch version only implemented `mean` aggregator, whereas the DGL version can use `mean`, `gcn`, `pool`, `lstm`, thanks to the SAGEConv in DGL library.

```
>>> sparse_col = ["sex", "occupation", "genre1", "genre2", "genre3"]
>>> dense_col = ["age"]
>>> user_col = ["sex", "age", "occupation"]
>>> item_col = ["genre1", "genre2", "genre3"]
>>> train_data, data_info = DatasetFeat.build_trainset(train, user_col, item_col, sparse_
↪col, dense_col)

>>> from libreco.algorithms import GraphSage, GraphSageDGL
>>> model = GraphSage(  # PyTorch version
```

(continues on next page)

(continued from previous page)

```
        task,
        data_info,
        loss_type="cross_entropy",  # or "focal", "bpr", "max_margin"
        paradigm="u2i",  # or "i2i"
    )
>>> model = GraphSageDGL(  # DGL version
        task,
        data_info,
        loss_type="focal",
        paradigm="i2i",
        aggregator_type="mean",  # or "gcn", "pool", "lstm"
    )
>>> model.fit(train_data)
```

1.13 Data

1.13.1 Dataset

Classes for Transforming and Building Data.

class libreco.data.dataset.**DatasetPure**

Bases: `_Dataset`

Dataset class used for building pure collaborative filtering data.

Examples

```
>>> from libreco.data import DatasetPure
>>> train_data, data_info = DatasetPure.build_trainset(train_data)
>>> eval_data = DatasetPure.build_evalset(eval_data)
>>> test_data = DatasetPure.build_testset(test_data)
```

---

classmethod build_trainset(*train_data*, *shuffle=False*, *seed=42*)

Build transformed train data and data_info from original data.

Changed in version 1.0.0: Data construction in *Model Retrain* has been moved to *merge_trainset()*

Parameters

- **train_data** (*pandas.DataFrame*) – Data must contain at least three columns, i.e. `user`, `item`, `label`.
- **shuffle** (*bool, default: False*) – Whether to fully shuffle data.

  Warning: If your data is order or time dependent, it is not recommended to shuffle data.

- **seed** (*int, default: 42*) – Random seed.

Returns

- **trainset** (*TransformedSet*) – Transformed Data object used for training.
- **data_info** (*DataInfo*) – Object that contains some useful information.

classmethod merge_trainset(*train_data*, *data_info*, *merge_behavior=True*, *shuffle=False*, *seed=42*)

Build transformed data by merging new train data with old data.

New in version 1.0.0.

Parameters

- **train_data** (*pandas.DataFrame*) – Data must contain at least three columns, i.e. `user`, `item`, `label`.
- **data_info** (DataInfo) – Object that contains past data information.
- **merge_behavior** (*bool, default: True*) – Whether to merge the user behavior in old and new data.
- **shuffle** (*bool, default: False*) – Whether to fully shuffle data.
- **seed** (*int, default: 42*) – Random seed.

Returns

Transformed Data object used for training.

Return type

TransformedSet

classmethod build_evalset(*eval_data*, *shuffle=False*, *seed=42*)

Build transformed eval data from original data.

Changed in version 1.0.0: Data construction in *Model Retrain* has been moved to *merge_evalset()*

Parameters

- **eval_data** (*pandas.DataFrame*) – Data must contain at least two columns, i.e. *user*, *item*.
- **shuffle** (*bool, default: False*) – Whether to fully shuffle data.
- **seed** (*int, default: 42*) – Random seed.

Returns

Transformed Data object used for evaluating.

Return type

TransformedSet

classmethod build_testset(*test_data*, *shuffle=False*, *seed=42*)

Build transformed test data from original data.

Changed in version 1.0.0: Data construction in *Model Retrain* has been moved to *merge_testset()*

Parameters

- **test_data** (*pandas.DataFrame*) – Data must contain at least two columns, i.e. *user*, *item*.
- **shuffle** (*bool, default: False*) – Whether to fully shuffle data.
- **seed** (*int, default: 42*) – Random seed.

Returns

Transformed Data object used for testing.

Return type
TransformedSet

classmethod merge_evalset(*eval_data*, *data_info*, *shuffle=False*, *seed=42*)

Build transformed data by merging new train data with old data.

New in version 1.0.0.

Parameters

- **eval_data** (*pandas.DataFrame*) – Data must contain at least two columns, i.e. *user*, *item*.
- **data_info** (*DataInfo*) – Object that contains past data information.
- **shuffle** (*bool, default: False*) – Whether to fully shuffle data.
- **seed** (*int, default: 42*) – Random seed.

Returns
Transformed Data object used for testing.

Return type
TransformedSet

classmethod merge_testset(*test_data*, *data_info*, *shuffle=False*, *seed=42*)

Build transformed data by merging new train data with old data.

New in version 1.0.0.

Parameters

- **test_data** (*pandas.DataFrame*) – Data must contain at least two columns, i.e. *user*, *item*.
- **data_info** (*DataInfo*) – Object that contains past data information.
- **shuffle** (*bool, default: False*) – Whether to fully shuffle data.
- **seed** (*int, default: 42*) – Random seed.

Returns
Transformed Data object used for testing.

Return type
TransformedSet

static shuffle_data(*data*, *seed*)

Shuffle data randomly.

Parameters

- **data** (*pandas.DataFrame*) – Data to shuffle.
- **seed** (*int*) – Random seed.

Returns
Shuffled data.

Return type
pandas.DataFrame

class libreco.data.dataset.**DatasetFeat**

Bases: _Dataset

Dataset class used for building data contains features.

Examples

```
>>> from libreco.data import DatasetFeat
>>> train_data, data_info = DatasetFeat.build_trainset(train_data)
>>> eval_data = DatasetFeat.build_evalset(eval_data)
>>> test_data = DatasetFeat.build_testset(test_data)
```

---

classmethod build_trainset(*train_data*, *user_col=None*, *item_col=None*, *sparse_col=None*, *dense_col=None*, *multi_sparse_col=None*, *unique_feat=False*, *pad_val='missing'*, *shuffle=False*, *seed=42*)

Build transformed feat train data and data_info from original data.

Changed in version 1.0.0: Data construction in *Model Retrain* has been moved to *merge_trainset()*

Parameters

- **train_data** (*pandas.DataFrame*) – Data must contain at least three columns, i.e. `user`, `item`, `label`.
- **user_col** (*list of str or None, default: None*) – List of user feature column names.
- **item_col** (*list of str or None, default: None*) – List of item feature column names.
- **sparse_col** (*list of str or None, default: None*) – List of sparse feature columns names.
- **multi_sparse_col** (*nested lists of str or None, default: None*) – Nested lists of multi_sparse feature columns names. For example, `[["a", "b", "c"], ["d", "e"]]`
- **dense_col** (*list of str or None, default: None*) – List of dense feature column names.
- **unique_feat** (*bool, default: False*) – Whether the features of users and items are unique in train data.
- **pad_val** (*int or str or list, default: "missing"*) – Padding value in multi_sparse columns to ensure same length of all samples.
- **shuffle** (*bool, default: False*) – Whether to fully shuffle data.

  Warning: If your data is order or time dependent, it is not recommended to shuffle data.

- **seed** (*int, default: 42*) – Random seed.

Returns

- **trainset** (*TransformedSet*) – Transformed Data object used for training.
- **data_info** (*DataInfo*) – Object that contains some useful information.

classmethod merge_trainset(*train_data*, *data_info*, *merge_behavior=True*, *shuffle=False*, *seed=42*)

Build transformed data by merging new train data with old data.

New in version 1.0.0.

Parameters

- **train_data** (*pandas.DataFrame*) – Data must contain at least three columns, i.e. `user`, `item`, `label`.
- **data_info** (*DataInfo*) – Object that contains past data information.
- **merge_behavior** (*bool, default: True*) – Whether to merge the user behavior in old and new data.
- **shuffle** (*bool, default: False*) – Whether to fully shuffle data.
- **seed** (*int, default: 42*) – Random seed.

Returns
Transformed Data object used for training.

Return type
TransformedSet

classmethod build_evalset(*eval_data*, *shuffle=False*, *seed=42*)

Build transformed eval data from original data.

Changed in version 1.0.0: Data construction in *Model Retrain* has been moved to *merge_evalset()*

Parameters

- **eval_data** (*pandas.DataFrame*) – Data must contain at least two columns, i.e. *user*, *item*.
- **shuffle** (*bool, default: False*) – Whether to fully shuffle data.
- **seed** (*int, default: 42*) – Random seed.

Returns
Transformed Data object used for evaluating.

Return type
TransformedSet

classmethod build_testset(*test_data*, *shuffle=False*, *seed=42*)

Build transformed test data from original data.

Changed in version 1.0.0: Data construction in *Model Retrain* has been moved to *merge_testset()*

Parameters

- **test_data** (*pandas.DataFrame*) – Data must contain at least two columns, i.e. *user*, *item*.
- **shuffle** (*bool, default: False*) – Whether to fully shuffle data.
- **seed** (*int, default: 42*) – Random seed.

Returns
Transformed Data object used for testing.

Return type
TransformedSet

classmethod merge_evalset(*eval_data*, *data_info*, *shuffle=False*, *seed=42*)

Build transformed data by merging new train data with old data.

New in version 1.0.0.

Parameters

- **eval_data** (*pandas.DataFrame*) – Data must contain at least two columns, i.e. *user*, *item*.
- **data_info** (DataInfo) – Object that contains past data information.
- **shuffle** (*bool, default: False*) – Whether to fully shuffle data.
- **seed** (*int, default: 42*) – Random seed.

Returns
Transformed Data object used for testing.

Return type
TransformedSet

classmethod merge_testset(*test_data*, *data_info*, *shuffle=False*, *seed=42*)

Build transformed data by merging new train data with old data.

New in version 1.0.0.

Parameters

- **test_data** (*pandas.DataFrame*) – Data must contain at least two columns, i.e. *user*, *item*.
- **data_info** (DataInfo) – Object that contains past data information.
- **shuffle** (*bool, default: False*) – Whether to fully shuffle data.
- **seed** (*int, default: 42*) – Random seed.

Returns
Transformed Data object used for testing.

Return type
TransformedSet

static shuffle_data(*data*, *seed*)

Shuffle data randomly.

Parameters

- **data** (*pandas.DataFrame*) – Data to shuffle.
- **seed** (*int*) – Random seed.

Returns
Shuffled data.

Return type
pandas.DataFrame

1.13.2 DataInfo

class libreco.data.**DataInfo**(*col_name_mapping=None*, *interaction_data=None*, *user_sparse_unique=None*, *user_dense_unique=None*, *item_sparse_unique=None*, *item_dense_unique=None*, *user_indices=None*, *item_indices=None*, *user_unique_vals=None*, *item_unique_vals=None*, *sparse_unique_vals=None*, *sparse_offset=None*, *sparse_oov=None*, *multi_sparse_unique_vals=None*, *multi_sparse_combine_info=None*)

Object for storing and updating indices and features information.

Parameters

- **col_name_mapping** (*dict of {dict : int} or None, default: None*) – Column name to index mapping, which has the format: `{column_family_name: {column_name: index}}`. If no such family, the default format would be: {column_family_name: {[]: []}
- **interaction_data** (*pandas.DataFrame or None, default: None*) – Data contains `user`, `item` and `label` columns
- **user_sparse_unique** (*numpy.ndarray or None, default: None*) – Unique sparse features for all users in train data.
- **user_dense_unique** (*numpy.ndarray or None, default: None*) – Unique dense features for all users in train data.
- **item_sparse_unique** (*numpy.ndarray or None, default: None*) – Unique sparse features for all items in train data.
- **item_dense_unique** (*numpy.ndarray or None, default: None*) – Unique dense features for all items in train data.
- **user_indices** (*numpy.ndarray or None, default: None*) – Mapped inner user indices from train data.
- **item_indices** (*numpy.ndarray or None, default: None*) – Mapped inner item indices from train data.
- **user_unique_vals** (*numpy.ndarray or None, default: None*) – All the unique users in train data.
- **item_unique_vals** (*numpy.ndarray or None, default: None*) – All the unique items in train data.
- **sparse_unique_vals** (*dict of {str : numpy.ndarray} or None, default: None*) – All sparse features' unique values.
- **sparse_offset** (*numpy.ndarray or None, default: None*) – Offset for each sparse feature in all sparse values. Often used in the `embedding` layer.
- **sparse_oov** (*numpy.ndarray or None, default: None*) – Out-of-vocabulary place for each sparse feature. Often used in cold-start.
- **multi_sparse_unique_vals** (*dict of {str : numpy.ndarray} or None, default: None*) – All multi-sparse features' unique values.
- **multi_sparse_combine_info** (*MultiSparseInfo or None, default: None*) – Multi-sparse field information.

Variables

- **col_name_mapping** (*dict of {dict : int} or None*) – See Parameters

- **user_consumed** (*dict of {int, list}*) – Every users' consumed items in train data.
- **item_consumed** (*dict of {int, list}*) – Every items' consumed users in train data.
- **popular_items** (*list*) – A number of popular items in train data. Often used in cold-start.

See also:

MultiSparseInfo

property global_mean

Mean value of all labels in *rating* task.

property min_max_rating

Min and max value of all labels in *rating* task.

property sparse_col

Sparse column name to index mapping.

property dense_col

Dense column name to index mapping.

property user_sparse_col

User sparse column name to index mapping.

property user_dense_col

User dense column name to index mapping.

property item_sparse_col

Item sparse column name to index mapping.

property item_dense_col

Item dense column name to index mapping.

property user_col

All the user column names, including sparse and dense.

property item_col

All the item column names, including sparse and dense.

property n_users

Number of users in train data.

property n_items

Number of items in train data.

property user2id

User original id to inner id mapping.

property item2id

Item original id to inner id mapping.

property id2user

User inner id to original id mapping.

property id2item

User inner id to original id mapping.

property data_size

Train data size.

__repr__()

Output train data information: "n_users, n_items, data density".

assign_user_features(*user_data*)

Assign user features to this `data_info` object from `user_data`.

Parameters

user_data (*pandas.DataFrame*) – Data contains new user features.

assign_item_features(*item_data*)

Assign item features to this `data_info` object from `item_data`.

Parameters

item_data (*pandas.DataFrame*) – Data contains new item features.

save(*path*, *model_name*)

Save *DataInfo* Object.

Parameters

- **path** (*str*) – File folder path to save *DataInfo*.
- **model_name** (*str*) – Name of the saved file.

classmethod load(*path*, *model_name*)

Load saved *DataInfo*.

Parameters

- **path** (*str*) – File folder path to save *DataInfo*.
- **model_name** (*str*) – Name of the saved file.

class libreco.data.**MultiSparseInfo**(*field_offset: Iterable[int]*, *field_len: Iterable[int]*, *feat_oov: ndarray*)

`dataclass` object for storing Multi-sparse features information.

A group of multi-sparse features are considered a "field". e.g. ["genre1", "genre2", "genre3"] form a field "genre". So this object contains fields' offset, field's length and fields' oov. Since features belong to the same field share one oov.

Variables

- **field_offset** (*list of int*) – All multi-sparse fields' offset in all expanded sparse features.
- **field_len** (*list of int*) – All multi-sparse fields' sizes.
- **feat_oov** (*numpy.ndarray*) – All multi-sparse fields' oov.

1.13.3 Split

libreco.data.**random_split**(*data*, *shuffle=True*, *test_size=None*, *multi_ratios=None*, *filter_unknown=True*, *pad_unknown=False*, *pad_val=None*, *seed=42*)

Split the data randomly.

Parameters

- **data** (*pandas.DataFrame*) – The data to split.
- **shuffle** (*bool, default: True*) – Whether to shuffle data when splitting.
- **test_size** (*float or None, default: None*) – Test data ratio.

- **multi_ratios** (*list of float, tuple of (float,) or None, default: None*) – Ratios for splitting data in multiple parts. If `test_size` is not None, `multi_ratios` will be ignored.
- **filter_unknown** (*bool, default: True*) – Whether to filter out users and items that don't appear in the train data from eval and test data. Since models can only recommend items in the train data.
- **pad_unknown** (*bool, default: False*) – Fill the unknown users/items with `pad_val`. If `filter_unknown` is True, this parameter will be ignored.
- **pad_val** (*any, default: None*) – Pad value used in `pad_unknown`.
- **seed** (*int, default: 42*) – Random seed.

Returns
multiple data – The split data.

Return type
list of pandas.DataFrame

Raises
ValueError – If neither *test_size* nor *multi_ratio* is provided.

Examples

```
>>> train, test = random_split(data, test_size=0.2)
>>> train_data, eval_data, test_data = random_split(data, multi_ratios=[0.8, 0.1, 0.
↪1])
```

libreco.data.**split_by_ratio**(*data*, *order=True*, *shuffle=False*, *test_size=None*, *multi_ratios=None*, *filter_unknown=True*, *pad_unknown=False*, *pad_val=None*, *seed=42*)

Assign certain ratio of items to test data for each user.

Note: If a user's total # of interacted items is less than 3, these items will all been assigned to train data.

Parameters

- **data** (*pandas.DataFrame*) – The data to split.
- **order** (*bool, default: True*) – Whether to preserve order for user's item sequence.
- **shuffle** (*bool, default: False*) – Whether to shuffle data after splitting.
- **test_size** (*float or None, default: None*) – Test data ratio.
- **multi_ratios** (*list of float, tuple of (float,) or None, default: None*) – Ratios for splitting data in multiple parts. If `test_size` is not None, `multi_ratios` will be ignored.
- **filter_unknown** (*bool, default: True*) – Whether to filter out users and items that don't appear in the train data from eval and test data. Since models can only recommend items in the train data.
- **pad_unknown** (*bool, default: False*) – Fill the unknown users/items with `pad_val`. If `filter_unknown` is True, this parameter will be ignored.
- **pad_val** (*any, default: None*) – Pad value used in `pad_unknown`.

- **seed** (*int, default: 42*) – Random seed.

Returns
multiple data – The split data.

Return type
list of pandas.DataFrame

Raises
ValueError – If neither *test_size* nor *multi_ratio* is provided.

See also:

split_by_ratio_chrono

libreco.data.**split_by_num**(*data*, *order=True*, *shuffle=False*, *test_size=1*, *filter_unknown=True*, *pad_unknown=False*, *pad_val=None*, *seed=42*)

Assign a certain number of items to test data for each user.

Note: If a user's total # of interacted items is less than 3, these items will all been assigned to train data.

Parameters

- **data** (*pandas.DataFrame*) – The data to split.
- **order** (*bool, default: True*) – Whether to preserve order for user's item sequence.
- **shuffle** (*bool, default: False*) – Whether to shuffle data after splitting.
- **test_size** (*float or None, default: None*) – Test data ratio.
- **filter_unknown** (*bool, default: True*) – Whether to filter out users and items that don't appear in the train data from eval and test data. Since models can only recommend items in the train data.
- **pad_unknown** (*bool, default: False*) – Fill the unknown users/items with `pad_val`. If `filter_unknown` is True, this parameter will be ignored.
- **pad_val** (*any, default: None*) – Pad value used in `pad_unknown`.
- **seed** (*int, default: 42*) – Random seed.

Returns
multiple data – The split data.

Return type
list of pandas.DataFrame

Raises
ValueError – If neither *test_size* nor *multi_ratio* is provided.

See also:

split_by_num_chrono

libreco.data.**split_by_ratio_chrono**(*data*, *order=True*, *shuffle=False*, *test_size=None*, *multi_ratios=None*, *seed=42*)

Assign a certain ratio of items to test data for each user, where items are sorted by time first.

Important: This function implies the data should contain a **time** column.

Parameters

- **data** (*pandas.DataFrame*) – The data to split.
- **order** (*bool, default: True*) – Whether to preserve order for user's item sequence.
- **shuffle** (*bool, default: False*) – Whether to shuffle data after splitting.
- **test_size** (*float or None, default: None*) – Test data ratio.
- **multi_ratios** (*list of float, tuple of (float,) or None, default: None*) – Ratios for splitting data in multiple parts. If `test_size` is not None, `multi_ratios` will be ignored.
- **seed** (*int, default: 42*) – Random seed.

Returns
multiple data – The split data.

Return type
list of pandas.DataFrame

Raises
ValueError – If neither *test_size* nor *multi_ratio* is provided.

See also:

split_by_ratio

libreco.data.**split_by_num_chrono**(*data*, *order=True*, *shuffle=False*, *test_size=1*, *seed=42*)

Assign a certain number of items to test data for each user, where items are sorted by time first.

Important: This function implies the data should contain a **time** column.

Parameters

- **data** (*pandas.DataFrame*) – The data to split.
- **order** (*bool, default: True*) – Whether to preserve order for user's item sequence.
- **shuffle** (*bool, default: False*) – Whether to shuffle data after splitting.
- **test_size** (*float or None, default: None*) – Test data ratio.
- **seed** (*int, default: 42*) – Random seed.

Returns
multiple data – The split data.

Return type
list of pandas.DataFrame

Raises
ValueError – If neither *test_size* nor *multi_ratio* is provided.

See also:

split_by_num

1.13.4 TransformedSet

class libreco.data.**TransformedSet**(*user_indices=None*, *item_indices=None*, *labels=None*, *sparse_indices=None*, *dense_values=None*, *train=True*)

Dataset after transforming.

Often generated by calling functions in `DatasetPure` or `DatasetFeat`, then `TransformedSet` is used in formal training.

Parameters

- **user_indices** (*numpy.ndarray or None, default: None*) – All user rows in data, represented in inner id.
- **item_indices** (*numpy.ndarray or None, default: None*) – All item rows in data, represented in inner id.
- **labels** (*numpy.ndarray or None, default: None*) – All labels in data.
- **sparse_indices** (*numpy.ndarray or None, default: None*) – All sparse rows in data, represented in inner id.
- **dense_values** (*numpy.ndarray or None, default: None*) – All dense rows in data.
- **train** (*bool, default: True*) – Whether it is train data.

See also:

DatasetPure, *DatasetFeat*

build_negative_samples(*data_info*, *num_neg=1*, *item_gen_mode='random'*, *seed=42*)

Perform negative sampling on all the data contained.

Parameters

- **data_info** (DataInfo) – Object contains data information.
- **num_neg** (*int, default: 1*) – Number of negative samples for each positive sample.
- **item_gen_mode** (*str, default: 'random'*) – Sampling strategy, currently only 'random' is supported.
- **seed** (*int, default: 42*) – Random seed.

property user_indices

All user rows in data

property item_indices

All item rows in data

property sparse_indices

All sparse rows in data

property dense_values

All dense rows in data

property labels

All labels in data

property sparse_interaction

User-item interaction data, in `scipy.sparse.csr_matrix` format.

1.14 Algorithms

1.14.1 Base Classes

class libreco.bases.**Base**(*task*, *data_info*, *lower_upper_bound=None*)

Bases: ABC

Base class for all recommendation models.

Parameters

- **task** ({'rating', 'ranking'}) – Recommendation task. See *Task*.
- **data_info** (*DataInfo* object) – Object that contains useful information for training and inference.
- **lower_upper_bound** (*list or tuple, default: None*) – Lower and upper score bound for rating task.

abstract fit(*train_data*, ***kwargs*)

Fit model on the training data.

Parameters
train_data (*TransformedSet* object) – Data object used for training.

abstract predict(*user*, *item*, ***kwargs*)

Predict score for given user and item.

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids.
- **item** (*int or str or array_like*) – Item id or batch of item ids.

Returns
prediction – Predicted scores for each user-item pair.

Return type
float or numpy.ndarray

abstract recommend_user(*user*, *n_rec*, ***kwargs*)

Recommend a list of items for given user.

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids to recommend.
- **n_rec** (*int*) – Number of recommendations to return.

Returns
recommendation – Recommendation result with user ids as keys and array_like recommended items as values.

Return type
dict of {Union[int, str, array_like] : numpy.ndarray}

abstract save(*path*, *model_name*, ***kwargs*)

Save model for inference or retraining.

Parameters

- **path** (*str*) – File folder path to save model.

- **model_name** (*str*) – Name of the saved model file.

See also:

load

abstract classmethod load(*path*, *model_name*, *data_info*, ***kwargs*)

Load saved model for inference.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.
- **data_info** (*DataInfo* object) – Object that contains some useful information.

Returns
model – Loaded model.

Return type
type(cls)

See also:

save

class libreco.bases.**EmbedBase**(*task*, *data_info*, *embed_size*, *lower_upper_bound=None*)

Bases: *Base*

Base class for embed models.

Models that can generate user and item embeddings.

Parameters

- **task** (*{'rating', 'ranking'}*) – Recommendation task. See *Task*.
- **data_info** (*DataInfo* object) – Object that contains useful information for training and inference.
- **embed_size** (*int*) – Vector size of embeddings.
- **lower_upper_bound** (*tuple or None, default: None*) – Lower and upper score bound for *rating* task.

fit(*train_data*, *verbose=1*, *shuffle=True*, *eval_data=None*, *metrics=None*, *k=10*, *eval_batch_size=8192*, *eval_user_num=None*)

Fit embed model on the training data.

Parameters

- **train_data** (*TransformedSet* object) – Data object used for training.
- **verbose** (*int, default: 1*) – Print verbosity. If *eval_data* is provided, setting it to higher than 1 will print evaluation metrics during training.
- **shuffle** (*bool, default: True*) – Whether to shuffle the training data.
- **eval_data** (*TransformedSet* object, default: None) – Data object used for evaluating.
- **metrics** (*list or None, default: None*) – List of metrics for evaluating.
- **k** (*int, default: 10*) – Parameter of metrics, e.g. recall at k, ndcg at k
- **eval_batch_size** (*int, default: 8192*) – Batch size for evaluating.

- **eval_user_num** (*int or None, default: None*) – Number of users for evaluating. Setting it to a positive number will sample users randomly from eval data.

Raises
RuntimeError – If *fit()* is called from a loaded model(*load()*).

predict(*user*, *item*, *cold_start='average'*, *inner_id=False*)

Make prediction(s) on given user(s) and item(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids.
- **item** (*int or str or array_like*) – Item id or batch of item ids.
- **cold_start** (*{'popular', 'average'}, default: 'average'*) – Cold start strategy.
  - 'popular' will sample from popular items.
  - 'average' will use the average of all the user/item embeddings as the representation of the cold-start user/item.
- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.

Returns
prediction – Predicted scores for each user-item pair.

Return type
float or numpy.ndarray

recommend_user(*user*, *n_rec*, *cold_start='average'*, *inner_id=False*, *filter_consumed=True*, *random_rec=False*)

Recommend a list of items for given user(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids to recommend.
- **n_rec** (*int*) – Number of recommendations to return.
- **cold_start** (*{'popular', 'average'}, default: 'average'*) – Cold start strategy.
  - 'popular' will sample from popular items.
  - 'average' will use the average of all the user/item embeddings as the representation of the cold-start user/item.
- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.
- **filter_consumed** (*bool, default: True*) – Whether to filter out items that a user has previously consumed.
- **random_rec** (*bool, default: False*) – Whether to choose items for recommendation based on their prediction scores.

Returns
recommendation – Recommendation result with user ids as keys and array_like recommended items as values.

Return type
dict of {Union[int, str, array_like] : numpy.ndarray}

save(*path*, *model_name*, *inference_only=False*, ***kwargs*)

Save embed model for inference or retraining.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.
- **inference_only** (*bool, default: False*) – Whether to save model only for inference. If it is True, only embeddings will be saved. Otherwise, model variables will be saved.

See also:

load

classmethod load(*path*, *model_name*, *data_info*, ***kwargs*)

Load saved embed model for inference.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.
- **data_info** (*DataInfo* object) – Object that contains some useful information.

Returns

model – Loaded embed model.

Return type

type(cls)

See also:

save

get_user_embedding(*user=None*)

Get user embedding(s) from the model.

Parameters

user (*int or str or None*) – Query user id. If it is None, all user embeddings will be returned.

Returns

user_embedding – Returned user embeddings.

Return type

numpy.ndarray

Raises

- **ValueError** – If the user does not appear in the training data.
- **AssertionError** – If the model has not been trained.

get_item_embedding(*item=None*)

Get item embedding(s) from the model.

Parameters

item (*int or str or None*) – Query item id. If it is None, all item embeddings will be returned.

Returns

item_embedding – Returned item embeddings.

Return type
numpy.ndarray

Raises

- **ValueError** – If the item does not appear in the training data.
- **AssertionError** – If the model has not been trained.

init_knn(*approximate*, *sim_type*, *M=100*, *ef_construction=200*, *ef_search=200*)

Initialize k-nearest-search model.

Parameters

- **approximate** (*bool*) – Whether to use approximate nearest neighbor search. If it is True, nmslib must be installed. The *HNSW* method in *nmslib* is used.
- **sim_type** (*{'cosine', 'inner-product'}*) – Similarity space type.
- **M** (*int, default: 100*) – Parameter in *HNSW*, refer to nmslib doc.
- **ef_construction** (*int, default: 200*) – Parameter in *HNSW*, refer to nmslib doc.
- **ef_search** (*int, default: 200*) – Parameter in *HNSW*, refer to nmslib doc.

Raises

- **ValueError** – If sim_type is not one of ('cosine', 'inner-product').
- **ModuleNotFoundError** – If *approximate=True* and *nmslib* is not installed.

search_knn_users(*user*, *k*)

Search most similar k users.

Parameters

- **user** (*int or str*) – Query user id.
- **k** (*int*) – Number of similar users.

Returns
similar users – A list of k similar users.

Return type
list

search_knn_items(*item*, *k*)

Search most similar k items.

Parameters

- **item** (*int or str*) – Query item id.
- **k** (*int*) – Number of similar items.

Returns
similar items – A list of k similar items.

Return type
list

class libreco.bases.**TfBase**(*task*, *data_info*, *lower_upper_bound=None*, *tf_sess_config=None*)

Bases: *Base*

Base class for TF models.

Models that relies on TensorFlow graph for inference. Although some models such as *RNN4Rec*, *SVD* etc., are trained using TensorFlow, they don't belong to this base class since their inference only uses embeddings.

Parameters

- **task** (*{'rating', 'ranking'}*) – Recommendation task. See *Task*.
- **data_info** (*DataInfo* object) – Object that contains useful information for training and inference.
- **lower_upper_bound** (*tuple or None*) – Lower and upper score bound for *rating* task.
- **tf_sess_config** (*dict or None*) – Optional TensorFlow session config, see ConfigProto options.

fit(*train_data*, *verbose=1*, *shuffle=True*, *eval_data=None*, *metrics=None*, *k=10*, *eval_batch_size=8192*, *eval_user_num=None*)

Fit TF model on the training data.

Parameters

- **train_data** (*TransformedSet* object) – Data object used for training.
- **verbose** (*int, default: 1*) – Print verbosity. If *eval_data* is provided, setting it to higher than 1 will print evaluation metrics during training.
- **shuffle** (*bool, default: True*) – Whether to shuffle the training data.
- **eval_data** (*TransformedSet* object, default: None) – Data object used for evaluating.
- **metrics** (*list or None, default: None*) – List of metrics for evaluating.
- **k** (*int, default: 10*) – Parameter of metrics, e.g. recall at k, ndcg at k
- **eval_batch_size** (*int, default: 8192*) – Batch size for evaluating.
- **eval_user_num** (*int or None, default: None*) – Number of users for evaluating. Setting it to a positive number will sample users randomly from eval data.

Raises

RuntimeError – If *fit()* is called from a loaded model(*load()*).

predict(*user*, *item*, *feats=None*, *cold_start='average'*, *inner_id=False*)

Make prediction(s) on given user(s) and item(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids.
- **item** (*int or str or array_like*) – Item id or batch of item ids.
- **feats** (*dict or pandas.Series or None, default: None*) – Extra features used in prediction.
- **cold_start** (*{'popular', 'average'}, default: 'average'*) – Cold start strategy.
  - 'popular' will sample from popular items.
  - 'average' will use the average of all the user/item embeddings as the representation of the cold-start user/item.
- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.

Returns

prediction – Predicted scores for each user-item pair.

Return type
float or numpy.ndarray

recommend_user(*user*, *n_rec*, *user_feats=None*, *item_data=None*, *cold_start='average'*, *inner_id=False*, *filter_consumed=True*, *random_rec=False*)

Recommend a list of items for given user(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids to recommend.
- **n_rec** (*int*) – Number of recommendations to return.
- **user_feats** (*dict or pandas.Series or None, default: None*) – Extra user features for recommendation.
- **item_data** (*pandas.DataFrame or None, default: None*) – Extra item features for recommendation.
- **cold_start** (*{'popular', 'average'}, default: 'average'*) – Cold start strategy.
  - 'popular' will sample from popular items.
  - 'average' will use the average of all the user/item embeddings as the representation of the cold-start user/item.
- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.
- **filter_consumed** (*bool, default: True*) – Whether to filter out items that a user has previously consumed.
- **random_rec** (*bool, default: False*) – Whether to choose items for recommendation based on their prediction scores.

Returns
recommendation – Recommendation result with user ids as keys and array_like recommended items as values.

Return type
dict of {Union[int, str, array_like] : numpy.ndarray}

save(*path*, *model_name*, *manual=True*, *inference_only=False*)

Save TF model for inference or retraining.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.
- **manual** (*bool, default: True*) – Whether to save model variables using numpy.
- **inference_only** (*bool, default: False*) – Whether to save model variables only for inference.

See also:

load

classmethod load(*path*, *model_name*, *data_info*, *manual=True*)

Load saved TF model for inference.

Parameters

- **path** (*str*) – File folder path to save model.

- **model_name** (*str*) – Name of the saved model file.
- **data_info** (*DataInfo* object) – Object that contains some useful information.
- **manual** (*bool, default: True*) – Whether to load model variables using numpy. If you save the model using *manual*, you should also load the mode using *manual*.

Returns
model – Loaded TF model.

Return type
type(cls)

See also:

save

class libreco.bases.**CfBase**(*task*, *data_info*, *cf_type*, *sim_type='cosine'*, *k_sim=20*, *store_top_k=True*, *block_size=None*, *num_threads=1*, *min_common=1*, *mode='invert'*, *seed=42*, *lower_upper_bound=None*)

Bases: *Base*

Base class for CF models.

Parameters

- **task** (*{'rating', 'ranking'}*) – Recommendation task. See *Task*.
- **data_info** (*DataInfo* object) – Object that contains useful information for training and inference.
- **cf_type** (*{'user_cf', 'item_cf'}*) – Specific CF type.
- **sim_type** (*{'cosine', 'pearson', 'jaccard'}, default: 'cosine'*) – Types for computing similarities.
- **k_sim** (*int, default: 20*) – Number of similar items to use.
- **store_top_k** (*bool, default: True*) – Whether to store top k similar users after training.
- **block_size** (*int or None, default: None*) – Block size for computing similarity matrix. Large block size makes computation faster, but may cause memory issue.
- **num_threads** (*int, default: 1*) – Number of threads to use.
- **min_common** (*int, default: 1*) – Number of minimum common users to consider when computing similarities.
- **mode** (*{'forward', 'invert'}, default: 'invert'*) – Whether to use forward index or invert index.
- **seed** (*int, default: 42*) – Random seed.
- **lower_upper_bound** (*tuple or None, default: None*) – Lower and upper score bound for *rating* task.

fit(*train_data*, *verbose=1*, *eval_data=None*, *metrics=None*, *k=10*, *eval_batch_size=8192*, *eval_user_num=None*)

Fit CF model on the training data.

Parameters

- **train_data** (*TransformedSet* object) – Data object used for training.

- **verbose** (*int, default: 1*) – Print verbosity. If *eval_data* is provided, setting it to higher than 1 will print evaluation metrics during training.
- **eval_data** (*TransformedSet* object, default: None) – Data object used for evaluating.
- **metrics** (*list or None, default: None*) – List of metrics for evaluating.
- **k** (*int, default: 10*) – Parameter of metrics, e.g. recall at k, ndcg at k
- **eval_batch_size** (*int, default: 8192*) – Batch size for evaluating.
- **eval_user_num** (*int or None, default: None*) – Number of users for evaluating. Setting it to a positive number will sample users randomly from eval data.

recommend_user(*user*, *n_rec*, *cold_start='popular'*, *inner_id=False*, *filter_consumed=True*, *random_rec=False*)

Recommend a list of items for given user(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids to recommend.
- **n_rec** (*int*) – Number of recommendations to return.
- **cold_start** (*{'popular'}, default: 'popular'*) – Cold start strategy, CF models can only use 'popular' strategy.
- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.
- **filter_consumed** (*bool, default: True*) – Whether to filter out items that a user has previously consumed.
- **random_rec** (*bool, default: False*) – Whether to choose items for recommendation based on their prediction scores.

Returns

recommendation – Recommendation result with user ids as keys and array_like recommended items as values.

Return type

dict[Union[int, str, array_like], numpy.ndarray]

save(*path*, *model_name*, ***kwargs*)

Save model for inference or retraining.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.

See also:

load

classmethod load(*path*, *model_name*, *data_info*, ***kwargs*)

Load saved model for inference.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.
- **data_info** (*DataInfo* object) – Object that contains some useful information.

Returns
model – Loaded model.

Return type
type(cls)

See also:

save

abstract predict(*user*, *item*, ***kwargs*)

Predict score for given user and item.

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids.
- **item** (*int or str or array_like*) – Item id or batch of item ids.

Returns
prediction – Predicted scores for each user-item pair.

Return type
float or numpy.ndarray

class libreco.bases.**GensimBase**(*task*, *data_info*, *embed_size=16*, *norm_embed=False*, *window_size=5*, *n_epochs=5*, *n_threads=0*, *seed=42*, *lower_upper_bound=None*)

Bases: *EmbedBase*

fit(*train_data*, *verbose=1*, *shuffle=True*, *eval_data=None*, *metrics=None*, *k=10*, *eval_batch_size=8192*, *eval_user_num=None*)

Fit embed model on the training data.

Parameters

- **train_data** (*TransformedSet* object) – Data object used for training.
- **verbose** (*int, default: 1*) – Print verbosity. If *eval_data* is provided, setting it to higher than 1 will print evaluation metrics during training.
- **shuffle** (*bool, default: True*) – Whether to shuffle the training data.
- **eval_data** (*TransformedSet* object, default: None) – Data object used for evaluating.
- **metrics** (*list or None, default: None*) – List of metrics for evaluating.
- **k** (*int, default: 10*) – Parameter of metrics, e.g. recall at k, ndcg at k
- **eval_batch_size** (*int, default: 8192*) – Batch size for evaluating.
- **eval_user_num** (*int or None, default: None*) – Number of users for evaluating. Setting it to a positive number will sample users randomly from eval data.

Raises
RuntimeError – If *fit()* is called from a loaded model(*load()*).

save(*path*, *model_name*, *inference_only=False*, ***_*)

Save embed model for inference or retraining.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.

- **inference_only** (*bool, default: False*) – Whether to save model only for inference. If it is True, only embeddings will be saved. Otherwise, model variables will be saved.

See also:

load

rebuild_model(*path*, *model_name*)

Assign the saved model variables to the newly initialized model.

This method is used before retraining the new model, in order to avoid training from scratch every time we get some new data.

Parameters

- **path** (*str*) – File folder path for the saved model variables.
- **model_name** (*str*) – Name of the saved model file.

get_item_embedding(*item=None*)

Get item embedding(s) from the model.

Parameters
item (*int or str or None*) – Query item id. If it is None, all item embeddings will be returned.

Returns
item_embedding – Returned item embeddings.

Return type
numpy.ndarray

Raises

- **ValueError** – If the item does not appear in the training data.
- **AssertionError** – If the model has not been trained.

get_user_embedding(*user=None*)

Get user embedding(s) from the model.

Parameters
user (*int or str or None*) – Query user id. If it is None, all user embeddings will be returned.

Returns
user_embedding – Returned user embeddings.

Return type
numpy.ndarray

Raises

- **ValueError** – If the user does not appear in the training data.
- **AssertionError** – If the model has not been trained.

init_knn(*approximate*, *sim_type*, *M=100*, *ef_construction=200*, *ef_search=200*)

Initialize k-nearest-search model.

Parameters

- **approximate** (*bool*) – Whether to use approximate nearest neighbor search. If it is True, nmslib must be installed. The *HNSW* method in *nmslib* is used.

- **sim_type** (*{'cosine', 'inner-product'}*) – Similarity space type.
- **M** (*int, default: 100*) – Parameter in *HNSW*, refer to nmslib doc.
- **ef_construction** (*int, default: 200*) – Parameter in *HNSW*, refer to nmslib doc.
- **ef_search** (*int, default: 200*) – Parameter in *HNSW*, refer to nmslib doc.

Raises

- **ValueError** – If sim_type is not one of ('cosine', 'inner-product').
- **ModuleNotFoundError** – If *approximate=True* and *nmslib* is not installed.

classmethod load(*path*, *model_name*, *data_info*, ***kwargs*)

Load saved embed model for inference.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.
- **data_info** (*DataInfo* object) – Object that contains some useful information.

Returns
model – Loaded embed model.

Return type
type(cls)

See also:

save

predict(*user*, *item*, *cold_start='average'*, *inner_id=False*)

Make prediction(s) on given user(s) and item(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids.
- **item** (*int or str or array_like*) – Item id or batch of item ids.
- **cold_start** (*{'popular', 'average'}, default: 'average'*) – Cold start strategy.
  - 'popular' will sample from popular items.
  - 'average' will use the average of all the user/item embeddings as the representation of the cold-start user/item.
- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.

Returns
prediction – Predicted scores for each user-item pair.

Return type
float or numpy.ndarray

recommend_user(*user*, *n_rec*, *cold_start='average'*, *inner_id=False*, *filter_consumed=True*, *random_rec=False*)

Recommend a list of items for given user(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids to recommend.

- **n_rec** (*int*) – Number of recommendations to return.
- **cold_start** (*{'popular', 'average'}, default: 'average'*) – Cold start strategy.
  - 'popular' will sample from popular items.
  - 'average' will use the average of all the user/item embeddings as the representation of the cold-start user/item.
- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.
- **filter_consumed** (*bool, default: True*) – Whether to filter out items that a user has previously consumed.
- **random_rec** (*bool, default: False*) – Whether to choose items for recommendation based on their prediction scores.

Returns
recommendation – Recommendation result with user ids as keys and array_like recommended items as values.

Return type
dict of {Union[int, str, array_like] : numpy.ndarray}

search_knn_items(*item*, *k*)

Search most similar k items.

Parameters
- **item** (*int or str*) – Query item id.
- **k** (*int*) – Number of similar items.

Returns
similar items – A list of k similar items.

Return type
list

search_knn_users(*user*, *k*)

Search most similar k users.

Parameters
- **user** (*int or str*) – Query user id.
- **k** (*int*) – Number of similar users.

Returns
similar users – A list of k similar users.

Return type
list

1.14.2 UserCF

class libreco.algorithms.**UserCF**(*task*, *data_info*, *sim_type='cosine'*, *k_sim=20*, *store_top_k=True*, *block_size=None*, *num_threads=1*, *min_common=1*, *mode='invert'*, *seed=42*, *lower_upper_bound=None*)

Bases: *CfBase*

User Collaborative Filtering algorithm.

Parameters

- **task** (*{'rating', 'ranking'}*) – Recommendation task. See *Task*.
- **data_info** (*DataInfo* object) – Object that contains useful information for training and inference.
- **sim_type** (*{'cosine', 'pearson', 'jaccard'}, default: 'cosine'*) – Types for computing similarities.
- **k_sim** (*int, default: 20*) – Number of similar items to use.
- **store_top_k** (*bool, default: True*) – Whether to store top k similar users after training.
- **block_size** (*int or None, default: None*) – Block size for computing similarity matrix. Large block size makes computation faster, but may cause memory issue.
- **num_threads** (*int, default: 1*) – Number of threads to use.
- **min_common** (*int, default: 1*) – Number of minimum common users to consider when computing similarities.
- **mode** (*{'forward', 'invert'}, default: 'invert'*) – Whether to use forward index or invert index.
- **seed** (*int, default: 42*) – Random seed.
- **lower_upper_bound** (*tuple or None, default: None*) – Lower and upper score bound for *rating* task.

predict(*user*, *item*, *cold_start='popular'*, *inner_id=False*)

Make prediction(s) on given user(s) and item(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids.
- **item** (*int or str or array_like*) – Item id or batch of item ids.
- **cold_start** (*{'popular'}, default: 'popular'*) – Cold start strategy, ItemCF can only use 'popular' strategy.
- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.

Returns

prediction – Predicted scores for each user-item pair.

Return type

float or numpy.ndarray

fit(*train_data*, *verbose=1*, *eval_data=None*, *metrics=None*, *k=10*, *eval_batch_size=8192*, *eval_user_num=None*)

Fit CF model on the training data.

Parameters

- **train_data** (*TransformedSet* object) – Data object used for training.
- **verbose** (*int, default: 1*) – Print verbosity. If *eval_data* is provided, setting it to higher than 1 will print evaluation metrics during training.
- **eval_data** (*TransformedSet* object, default: None) – Data object used for evaluating.
- **metrics** (*list or None, default: None*) – List of metrics for evaluating.
- **k** (*int, default: 10*) – Parameter of metrics, e.g. recall at k, ndcg at k
- **eval_batch_size** (*int, default: 8192*) – Batch size for evaluating.
- **eval_user_num** (*int or None, default: None*) – Number of users for evaluating. Setting it to a positive number will sample users randomly from eval data.

classmethod load(*path*, *model_name*, *data_info*, ***kwargs*)

Load saved model for inference.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.
- **data_info** (*DataInfo* object) – Object that contains some useful information.

Returns

model – Loaded model.

Return type

type(cls)

See also:

save

recommend_user(*user*, *n_rec*, *cold_start='popular'*, *inner_id=False*, *filter_consumed=True*, *random_rec=False*)

Recommend a list of items for given user(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids to recommend.
- **n_rec** (*int*) – Number of recommendations to return.
- **cold_start** (*{'popular'}, default: 'popular'*) – Cold start strategy, CF models can only use 'popular' strategy.
- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.
- **filter_consumed** (*bool, default: True*) – Whether to filter out items that a user has previously consumed.
- **random_rec** (*bool, default: False*) – Whether to choose items for recommendation based on their prediction scores.

Returns

recommendation – Recommendation result with user ids as keys and array_like recommended items as values.

Return type
dict[Union[int, str, array_like], numpy.ndarray]

save(*path*, *model_name*, ***kwargs*)

Save model for inference or retraining.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.

See also:

load

1.14.3 ItemCF

class libreco.algorithms.**ItemCF**(*task*, *data_info*, *sim_type='cosine'*, *k_sim=20*, *store_top_k=True*, *block_size=None*, *num_threads=1*, *min_common=1*, *mode='invert'*, *seed=42*, *lower_upper_bound=None*)

Bases: *CfBase*

Item Collaborative Filtering algorithm.

Parameters

- **task** (*{'rating', 'ranking'}*) – Recommendation task. See *Task*.
- **data_info** (*DataInfo* object) – Object that contains useful information for training and inference.
- **sim_type** (*{'cosine', 'pearson', 'jaccard'}, default: 'cosine'*) – Types for computing similarities.
- **k_sim** (*int, default: 20*) – Number of similar items to use.
- **store_top_k** (*bool, default: True*) – Whether to store top k similar items after training.
- **block_size** (*int or None, default: None*) – Block size for computing similarity matrix. Large block size makes computation faster, but may cause memory issue.
- **num_threads** (*int, default: 1*) – Number of threads to use.
- **min_common** (*int, default: 1*) – Number of minimum common items to consider when computing similarities.
- **mode** (*{'forward', 'invert'}, default: 'invert'*) – Whether to use forward index or invert index.
- **seed** (*int, default: 42*) – Random seed.
- **lower_upper_bound** (*tuple or None, default: None*) – Lower and upper score bound for *rating* task.

predict(*user*, *item*, *cold_start='popular'*, *inner_id=False*)

Make prediction(s) on given user(s) and item(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids.
- **item** (*int or str or array_like*) – Item id or batch of item ids.

- **cold_start** (*{'popular'}, default: 'popular'*) – Cold start strategy, ItemCF can only use 'popular' strategy.
- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.

Returns
prediction – Predicted scores for each user-item pair.

Return type
float or array_like

fit(*train_data*, *verbose=1*, *eval_data=None*, *metrics=None*, *k=10*, *eval_batch_size=8192*, *eval_user_num=None*)

Fit CF model on the training data.

Parameters

- **train_data** (*TransformedSet* object) – Data object used for training.
- **verbose** (*int, default: 1*) – Print verbosity. If *eval_data* is provided, setting it to higher than 1 will print evaluation metrics during training.
- **eval_data** (*TransformedSet* object, default: None) – Data object used for evaluating.
- **metrics** (*list or None, default: None*) – List of metrics for evaluating.
- **k** (*int, default: 10*) – Parameter of metrics, e.g. recall at k, ndcg at k
- **eval_batch_size** (*int, default: 8192*) – Batch size for evaluating.
- **eval_user_num** (*int or None, default: None*) – Number of users for evaluating. Setting it to a positive number will sample users randomly from eval data.

classmethod load(*path*, *model_name*, *data_info*, ***kwargs*)

Load saved model for inference.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.
- **data_info** (*DataInfo* object) – Object that contains some useful information.

Returns
model – Loaded model.

Return type
type(cls)

See also:

save

recommend_user(*user*, *n_rec*, *cold_start='popular'*, *inner_id=False*, *filter_consumed=True*, *random_rec=False*)

Recommend a list of items for given user(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids to recommend.
- **n_rec** (*int*) – Number of recommendations to return.

- **cold_start** (*{'popular'}, default: 'popular'*) – Cold start strategy, CF models can only use 'popular' strategy.
- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.
- **filter_consumed** (*bool, default: True*) – Whether to filter out items that a user has previously consumed.
- **random_rec** (*bool, default: False*) – Whether to choose items for recommendation based on their prediction scores.

Returns

recommendation – Recommendation result with user ids as keys and array_like recommended items as values.

Return type

dict[Union[int, str, array_like], numpy.ndarray]

save(*path*, *model_name*, ***kwargs*)

Save model for inference or retraining.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.

See also:

load

1.14.4 SVD

class libreco.algorithms.**SVD**(*task*, *data_info*, *loss_type='cross_entropy'*, *embed_size=16*, *n_epochs=20*, *lr=0.001*, *lr_decay=False*, *epsilon=1e-05*, *reg=None*, *batch_size=256*, *num_neg=1*, *seed=42*, *lower_upper_bound=None*, *tf_sess_config=None*)

Bases: *EmbedBase*

Singular Value Decomposition algorithm.

Parameters

- **task** (*{'rating', 'ranking'}*) – Recommendation task. See *Task*.
- **data_info** (*DataInfo* object) – Object that contains useful information for training and inference.
- **loss_type** (*{'cross_entropy', 'focal'}, default: 'cross_entropy'*) – Loss for model training.
- **embed_size** (*int, default: 16*) – Vector size of embeddings.
- **n_epochs** (*int, default: 10*) – Number of epochs for training.
- **lr** (*float, default 0.001*) – Learning rate for training.
- **lr_decay** (*bool, default: False*) – Whether to use learning rate decay.
- **epsilon** (*float, default: 1e-5*) – A small constant added to the denominator to improve numerical stability in Adam optimizer. According to the official comment, default value of *1e-8* for *epsilon* is generally not good, so here we choose *1e-5*. Users can try tuning this hyperparameter if the training is unstable.

- **reg** (*float or None, default: None*) – Regularization parameter, must be non-negative or None.
- **batch_size** (*int, default: 256*) – Batch size for training.
- **num_neg** (*int, default: 1*) – Number of negative samples for each positive sample, only used in *ranking* task.
- **seed** (*int, default: 42*) – Random seed.
- **lower_upper_bound** (*tuple or None, default: None*) – Lower and upper score bound for *rating* task.
- **tf_sess_config** (*dict or None, default: None*) – Optional TensorFlow session config, see ConfigProto options.

References

Yehuda Koren Matrix Factorization Techniques for Recommender Systems.

---

fit(*train_data*, *verbose=1*, *shuffle=True*, *eval_data=None*, *metrics=None*, *k=10*, *eval_batch_size=8192*, *eval_user_num=None*)

Fit embed model on the training data.

Parameters

- **train_data** (*TransformedSet* object) – Data object used for training.
- **verbose** (*int, default: 1*) – Print verbosity. If *eval_data* is provided, setting it to higher than 1 will print evaluation metrics during training.
- **shuffle** (*bool, default: True*) – Whether to shuffle the training data.
- **eval_data** (*TransformedSet* object, default: None) – Data object used for evaluating.
- **metrics** (*list or None, default: None*) – List of metrics for evaluating.
- **k** (*int, default: 10*) – Parameter of metrics, e.g. recall at k, ndcg at k
- **eval_batch_size** (*int, default: 8192*) – Batch size for evaluating.
- **eval_user_num** (*int or None, default: None*) – Number of users for evaluating. Setting it to a positive number will sample users randomly from eval data.

Raises

RuntimeError – If *fit()* is called from a loaded model(*load()*).

get_item_embedding(*item=None*)

Get item embedding(s) from the model.

Parameters

item (*int or str or None*) – Query item id. If it is None, all item embeddings will be returned.

Returns

item_embedding – Returned item embeddings.

Return type

numpy.ndarray

Raises

- **ValueError** – If the item does not appear in the training data.

- **AssertionError** – If the model has not been trained.

get_user_embedding(*user=None*)

Get user embedding(s) from the model.

Parameters

user (*int or str or None*) – Query user id. If it is None, all user embeddings will be returned.

Returns

user_embedding – Returned user embeddings.

Return type

numpy.ndarray

Raises

- **ValueError** – If the user does not appear in the training data.
- **AssertionError** – If the model has not been trained.

init_knn(*approximate*, *sim_type*, *M=100*, *ef_construction=200*, *ef_search=200*)

Initialize k-nearest-search model.

Parameters

- **approximate** (*bool*) – Whether to use approximate nearest neighbor search. If it is True, nmslib must be installed. The *HNSW* method in *nmslib* is used.
- **sim_type** (*{'cosine', 'inner-product'}*) – Similarity space type.
- **M** (*int, default: 100*) – Parameter in *HNSW*, refer to nmslib doc.
- **ef_construction** (*int, default: 200*) – Parameter in *HNSW*, refer to nmslib doc.
- **ef_search** (*int, default: 200*) – Parameter in *HNSW*, refer to nmslib doc.

Raises

- **ValueError** – If sim_type is not one of ('cosine', 'inner-product').
- **ModuleNotFoundError** – If *approximate=True* and *nmslib* is not installed.

classmethod load(*path*, *model_name*, *data_info*, ***kwargs*)

Load saved embed model for inference.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.
- **data_info** (*DataInfo* object) – Object that contains some useful information.

Returns

model – Loaded embed model.

Return type

type(cls)

See also:

save

predict(*user*, *item*, *cold_start='average'*, *inner_id=False*)

Make prediction(s) on given user(s) and item(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids.
- **item** (*int or str or array_like*) – Item id or batch of item ids.
- **cold_start** (*{'popular', 'average'}, default: 'average'*) – Cold start strategy.
  - 'popular' will sample from popular items.
  - 'average' will use the average of all the user/item embeddings as the representation of the cold-start user/item.
- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.

Returns

prediction – Predicted scores for each user-item pair.

Return type

float or numpy.ndarray

rebuild_model(*path*, *model_name*, *full_assign=True*)

Assign the saved model variables to the newly initialized model.

This method is used before retraining the new model, in order to avoid training from scratch every time we get some new data.

Parameters

- **path** (*str*) – File folder path for the saved model variables.
- **model_name** (*str*) – Name of the saved model file.
- **full_assign** (*bool, default: True*) – Whether to also restore the variables of Adam optimizer.

recommend_user(*user*, *n_rec*, *cold_start='average'*, *inner_id=False*, *filter_consumed=True*, *random_rec=False*)

Recommend a list of items for given user(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids to recommend.
- **n_rec** (*int*) – Number of recommendations to return.
- **cold_start** (*{'popular', 'average'}, default: 'average'*) – Cold start strategy.
  - 'popular' will sample from popular items.
  - 'average' will use the average of all the user/item embeddings as the representation of the cold-start user/item.
- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.
- **filter_consumed** (*bool, default: True*) – Whether to filter out items that a user has previously consumed.
- **random_rec** (*bool, default: False*) – Whether to choose items for recommendation based on their prediction scores.

Returns
recommendation – Recommendation result with user ids as keys and array_like recommended items as values.

Return type
dict of {Union[int, str, array_like] : numpy.ndarray}

save(*path*, *model_name*, *inference_only=False*, ***kwargs*)

Save embed model for inference or retraining.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.
- **inference_only** (*bool, default: False*) – Whether to save model only for inference. If it is True, only embeddings will be saved. Otherwise, model variables will be saved.

See also:

load

search_knn_items(*item*, *k*)

Search most similar k items.

Parameters

- **item** (*int or str*) – Query item id.
- **k** (*int*) – Number of similar items.

Returns
similar items – A list of k similar items.

Return type
list

search_knn_users(*user*, *k*)

Search most similar k users.

Parameters

- **user** (*int or str*) – Query user id.
- **k** (*int*) – Number of similar users.

Returns
similar users – A list of k similar users.

Return type
list

1.14.5 SVD++

class libreco.algorithms.**SVDpp**(*task*, *data_info*, *loss_type='cross_entropy'*, *embed_size=16*, *n_epochs=20*, *lr=0.001*, *lr_decay=False*, *epsilon=1e-05*, *reg=None*, *batch_size=256*, *num_neg=1*, *seed=42*, *recent_num=30*, *lower_upper_bound=None*, *tf_sess_config=None*)

Bases: *EmbedBase*

SVD++ algorithm.

Parameters

- **task** (*{'rating', 'ranking'}*) – Recommendation task. See *Task*.
- **data_info** (*DataInfo* object) – Object that contains useful information for training and inference.
- **loss_type** (*{'cross_entropy', 'focal'}, default: 'cross_entropy'*) – Loss for model training.
- **embed_size** (*int, default: 16*) – Vector size of embeddings.
- **n_epochs** (*int, default: 10*) – Number of epochs for training.
- **lr** (*float, default 0.001*) – Learning rate for training.
- **lr_decay** (*bool, default: False*) – Whether to use learning rate decay.
- **epsilon** (*float, default: 1e-5*) – A small constant added to the denominator to improve numerical stability in Adam optimizer. According to the official comment, default value of *1e-8* for *epsilon* is generally not good, so here we choose *1e-5*. Users can try tuning this hyperparameter if the training is unstable.
- **reg** (*float or None, default: None*) – Regularization parameter, must be non-negative or None.
- **batch_size** (*int, default: 256*) – Batch size for training.
- **num_neg** (*int, default: 1*) – Number of negative samples for each positive sample, only used in *ranking* task.
- **seed** (*int, default: 42*) – Random seed.
- **lower_upper_bound** (*tuple or None, default: None*) – Lower and upper score bound for *rating* task.
- **tf_sess_config** (*dict or None, default: None*) – Optional TensorFlow session config, see ConfigProto options.

References

Yehuda Koren Factorization Meets the Neighborhood: a Multifaceted Collaborative Filtering Model.

fit(*train_data*, *verbose=1*, *shuffle=True*, *eval_data=None*, *metrics=None*, ***kwargs*)

Fit embed model on the training data.

Parameters

- **train_data** (*TransformedSet* object) – Data object used for training.
- **verbose** (*int, default: 1*) – Print verbosity. If *eval_data* is provided, setting it to higher than 1 will print evaluation metrics during training.

- **shuffle** (*bool, default: True*) – Whether to shuffle the training data.
- **eval_data** (*TransformedSet* object, default: None) – Data object used for evaluating.
- **metrics** (*list or None, default: None*) – List of metrics for evaluating.
- **k** (*int, default: 10*) – Parameter of metrics, e.g. recall at k, ndcg at k
- **eval_batch_size** (*int, default: 8192*) – Batch size for evaluating.
- **eval_user_num** (*int or None, default: None*) – Number of users for evaluating. Setting it to a positive number will sample users randomly from eval data.

Raises
RuntimeError – If *fit()* is called from a loaded model(*load()*).

get_item_embedding(*item=None*)

Get item embedding(s) from the model.

Parameters
item (*int or str or None*) – Query item id. If it is None, all item embeddings will be returned.

Returns
item_embedding – Returned item embeddings.

Return type
numpy.ndarray

Raises
- **ValueError** – If the item does not appear in the training data.
- **AssertionError** – If the model has not been trained.

get_user_embedding(*user=None*)

Get user embedding(s) from the model.

Parameters
user (*int or str or None*) – Query user id. If it is None, all user embeddings will be returned.

Returns
user_embedding – Returned user embeddings.

Return type
numpy.ndarray

Raises
- **ValueError** – If the user does not appear in the training data.
- **AssertionError** – If the model has not been trained.

init_knn(*approximate*, *sim_type*, *M=100*, *ef_construction=200*, *ef_search=200*)

Initialize k-nearest-search model.

Parameters
- **approximate** (*bool*) – Whether to use approximate nearest neighbor search. If it is True, nmslib must be installed. The *HNSW* method in *nmslib* is used.
- **sim_type** (*{'cosine', 'inner-product'}*) – Similarity space type.
- **M** (*int, default: 100*) – Parameter in *HNSW*, refer to nmslib doc.

- **ef_construction** (*int, default: 200*) – Parameter in *HNSW*, refer to nmslib doc.
- **ef_search** (*int, default: 200*) – Parameter in *HNSW*, refer to nmslib doc.

Raises

- **ValueError** – If sim_type is not one of ('cosine', 'inner-product').
- **ModuleNotFoundError** – If *approximate=True* and *nmslib* is not installed.

classmethod load(*path*, *model_name*, *data_info*, ***kwargs*)

Load saved embed model for inference.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.
- **data_info** (*DataInfo* object) – Object that contains some useful information.

Returns
model – Loaded embed model.

Return type
type(cls)

See also:

save

predict(*user*, *item*, *cold_start='average'*, *inner_id=False*)

Make prediction(s) on given user(s) and item(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids.
- **item** (*int or str or array_like*) – Item id or batch of item ids.
- **cold_start** (*{'popular', 'average'}, default: 'average'*) – Cold start strategy.
  - 'popular' will sample from popular items.
  - 'average' will use the average of all the user/item embeddings as the representation of the cold-start user/item.
- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.

Returns
prediction – Predicted scores for each user-item pair.

Return type
float or numpy.ndarray

recommend_user(*user*, *n_rec*, *cold_start='average'*, *inner_id=False*, *filter_consumed=True*, *random_rec=False*)

Recommend a list of items for given user(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids to recommend.
- **n_rec** (*int*) – Number of recommendations to return.
- **cold_start** (*{'popular', 'average'}, default: 'average'*) – Cold start strategy.

- 'popular' will sample from popular items.
- 'average' will use the average of all the user/item embeddings as the representation of the cold-start user/item.

- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.
- **filter_consumed** (*bool, default: True*) – Whether to filter out items that a user has previously consumed.
- **random_rec** (*bool, default: False*) – Whether to choose items for recommendation based on their prediction scores.

Returns
recommendation – Recommendation result with user ids as keys and array_like recommended items as values.

Return type
dict of {Union[int, str, array_like] : numpy.ndarray}

save(*path*, *model_name*, *inference_only=False*, ***kwargs*)

Save embed model for inference or retraining.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.
- **inference_only** (*bool, default: False*) – Whether to save model only for inference. If it is True, only embeddings will be saved. Otherwise, model variables will be saved.

See also:

load

search_knn_items(*item*, *k*)

Search most similar k items.

Parameters

- **item** (*int or str*) – Query item id.
- **k** (*int*) – Number of similar items.

Returns
similar items – A list of k similar items.

Return type
list

search_knn_users(*user*, *k*)

Search most similar k users.

Parameters

- **user** (*int or str*) – Query user id.
- **k** (*int*) – Number of similar users.

Returns
similar users – A list of k similar users.

Return type
list

1.14.6 ALS

class libreco.algorithms.**ALS**(*task*, *data_info*, *embed_size=16*, *n_epochs=10*, *reg=None*, *alpha=10*, *use_cg=True*, *n_threads=1*, *seed=42*, *lower_upper_bound=None*)

Bases: *EmbedBase*

Alternating Least Squares algorithm.

One can use conjugate gradient optimization and set more *n_threads* to accelerate training.

Parameters

- **task** (*{'rating', 'ranking'}*) – Recommendation task. See *Task*.
- **data_info** (*DataInfo* object) – Object that contains useful information for training and inference.
- **embed_size** (*int, default: 16*) – Vector size of embeddings.
- **n_epochs** (*int, default: 10*) – Number of epochs for training.
- **reg** (*float or None, default: None*) – Regularization parameter, must be non-negative or None.
- **alpha** (*int, default: 10*) – Parameter used for increasing confidence level, only applied for *ranking* task.
- **use_cg** (*bool, default: True*) – Whether to use *conjugate gradient* optimization. See reference.
- **n_threads** (*int, default: 1*) – Number of threads to use.
- **seed** (*int, default: 42*) – Random seed.
- **lower_upper_bound** (*tuple or None, default: None*) – Lower and upper score bound for *rating* task.

References

[1] *Haoming Li et al.* Matrix Completion via Alternating Least Square(ALS).

[2] *Yifan Hu et al.* Collaborative Filtering for Implicit Feedback Datasets.

[3] *Gábor Takács et al.* Applications of the Conjugate Gradient Method for Implicit Feedback Collaborative Filtering.

---

fit(*train_data*, *verbose=1*, *shuffle=True*, *eval_data=None*, *metrics=None*, *k=10*, *eval_batch_size=8192*, *eval_user_num=None*)

Fit ALS model on the training data.

Parameters

- **train_data** (*TransformedSet* object) – Data object used for training.
- **verbose** (*int, default: 1*) – Print verbosity. If *eval_data* is provided, setting it to higher than 1 will print evaluation metrics during training.
- **shuffle** (*bool, default: True*) – Whether to shuffle the training data.

- **eval_data** (*TransformedSet* object, default: None) – Data object used for evaluating.
- **metrics** (*list or None, default: None*) – List of metrics for evaluating.
- **k** (*int, default: 10*) – Parameter of metrics, e.g. recall at k, ndcg at k
- **eval_batch_size** (*int, default: 8192*) – Batch size for evaluating.
- **eval_user_num** (*int or None, default: None*) – Number of users for evaluating. Setting it to a positive number will sample users randomly from eval data.

save(*path*, *model_name*, ***kwargs*)

Save model for inference or retraining.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.

rebuild_model(*path*, *model_name*)

Reconstruct model for retraining.

Parameters

- **path** (*str*) – File folder path for saved model.
- **model_name** (*str*) – Name of the saved model file.

get_item_embedding(*item=None*)

Get item embedding(s) from the model.

Parameters
item (*int or str or None*) – Query item id. If it is None, all item embeddings will be returned.

Returns
item_embedding – Returned item embeddings.

Return type
numpy.ndarray

Raises

- **ValueError** – If the item does not appear in the training data.
- **AssertionError** – If the model has not been trained.

get_user_embedding(*user=None*)

Get user embedding(s) from the model.

Parameters
user (*int or str or None*) – Query user id. If it is None, all user embeddings will be returned.

Returns
user_embedding – Returned user embeddings.

Return type
numpy.ndarray

Raises

- **ValueError** – If the user does not appear in the training data.
- **AssertionError** – If the model has not been trained.

init_knn(*approximate*, *sim_type*, *M=100*, *ef_construction=200*, *ef_search=200*)

Initialize k-nearest-search model.

Parameters

- **approximate** (*bool*) – Whether to use approximate nearest neighbor search. If it is True, nmslib must be installed. The *HNSW* method in *nmslib* is used.
- **sim_type** (*{'cosine', 'inner-product'}*) – Similarity space type.
- **M** (*int, default: 100*) – Parameter in *HNSW*, refer to nmslib doc.
- **ef_construction** (*int, default: 200*) – Parameter in *HNSW*, refer to nmslib doc.
- **ef_search** (*int, default: 200*) – Parameter in *HNSW*, refer to nmslib doc.

Raises

- **ValueError** – If sim_type is not one of ('cosine', 'inner-product').
- **ModuleNotFoundError** – If *approximate=True* and *nmslib* is not installed.

classmethod load(*path*, *model_name*, *data_info*, ***kwargs*)

Load saved embed model for inference.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.
- **data_info** (*DataInfo* object) – Object that contains some useful information.

Returns

model – Loaded embed model.

Return type

type(cls)

See also:

save

predict(*user*, *item*, *cold_start='average'*, *inner_id=False*)

Make prediction(s) on given user(s) and item(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids.
- **item** (*int or str or array_like*) – Item id or batch of item ids.
- **cold_start** (*{'popular', 'average'}, default: 'average'*) – Cold start strategy.
  - 'popular' will sample from popular items.
  - 'average' will use the average of all the user/item embeddings as the representation of the cold-start user/item.
- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.

Returns

prediction – Predicted scores for each user-item pair.

Return type

float or numpy.ndarray

recommend_user(*user*, *n_rec*, *cold_start='average'*, *inner_id=False*, *filter_consumed=True*, *random_rec=False*)

Recommend a list of items for given user(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids to recommend.
- **n_rec** (*int*) – Number of recommendations to return.
- **cold_start** (*{'popular', 'average'}, default: 'average'*) – Cold start strategy.
  - 'popular' will sample from popular items.
  - 'average' will use the average of all the user/item embeddings as the representation of the cold-start user/item.
- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.
- **filter_consumed** (*bool, default: True*) – Whether to filter out items that a user has previously consumed.
- **random_rec** (*bool, default: False*) – Whether to choose items for recommendation based on their prediction scores.

Returns

recommendation – Recommendation result with user ids as keys and array_like recommended items as values.

Return type

dict of {Union[int, str, array_like] : numpy.ndarray}

search_knn_items(*item*, *k*)

Search most similar k items.

Parameters

- **item** (*int or str*) – Query item id.
- **k** (*int*) – Number of similar items.

Returns

similar items – A list of k similar items.

Return type

list

search_knn_users(*user*, *k*)

Search most similar k users.

Parameters

- **user** (*int or str*) – Query user id.
- **k** (*int*) – Number of similar users.

Returns

similar users – A list of k similar users.

Return type

list

1.14.7 NCF

class libreco.algorithms.**NCF**(*task*, *data_info*, *loss_type='cross_entropy'*, *embed_size=16*, *n_epochs=20*, *lr=0.01*, *lr_decay=False*, *epsilon=1e-05*, *reg=None*, *batch_size=256*, *num_neg=1*, *use_bn=True*, *dropout_rate=None*, *hidden_units=(128, 64, 32)*, *seed=42*, *lower_upper_bound=None*, *tf_sess_config=None*)

Bases: *TfBase*

Neural Collaborative Filtering algorithm.

Parameters

- **task** (*{'rating', 'ranking'}*) – Recommendation task. See *Task*.
- **data_info** (*DataInfo* object) – Object that contains useful information for training and inference.
- **loss_type** (*{'cross_entropy', 'focal'}, default: 'cross_entropy'*) – Loss for model training.
- **embed_size** (*int, default: 16*) – Vector size of embeddings.
- **n_epochs** (*int, default: 10*) – Number of epochs for training.
- **lr** (*float, default 0.001*) – Learning rate for training.
- **lr_decay** (*bool, default: False*) – Whether to use learning rate decay.
- **epsilon** (*float, default: 1e-5*) – A small constant added to the denominator to improve numerical stability in Adam optimizer. According to the official comment, default value of *1e-8* for *epsilon* is generally not good, so here we choose *1e-5*. Users can try tuning this hyperparameter if the training is unstable.
- **reg** (*float or None, default: None*) – Regularization parameter, must be non-negative or None.
- **batch_size** (*int, default: 256*) – Batch size for training.
- **num_neg** (*int, default: 1*) – Number of negative samples for each positive sample, only used in *ranking* task.
- **use_bn** (*bool, default: True*) – Whether to use batch normalization.
- **dropout_rate** (*float or None, default: None*) – Probability of an element to be zeroed. If it is None, dropout is not used.
- **hidden_units** (*int, list of int or tuple of (int,), default: (128, 64, 32)*) – Number of layers and corresponding layer size in MLP.

  Changed in version 1.0.0: Accept type of `int`, `list` or `tuple`, instead of `str`.
- **seed** (*int, default: 42*) – Random seed.
- **lower_upper_bound** (*tuple or None, default: None*) – Lower and upper score bound for *rating* task.
- **tf_sess_config** (*dict or None, default: None*) – Optional TensorFlow session config, see ConfigProto options.

References

Xiangnan He et al. Neural Collaborative Filtering.

predict(*user*, *item*, *feats=None*, *cold_start='average'*, *inner_id=False*)

Make prediction(s) on given user(s) and item(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids.
- **item** (*int or str or array_like*) – Item id or batch of item ids.
- **feats** (*None, default: None*) – NCF can't use features.
- **cold_start** (*{'popular', 'average'}, default: 'average'*) – Cold start strategy.
  - 'popular' will sample from popular items.
  - 'average' will use the average of all the user/item embeddings as the representation of the cold-start user/item.
- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.

Returns

prediction – Predicted scores for each user-item pair.

Return type

float or array_like

recommend_user(*user*, *n_rec*, *user_feats=None*, *item_data=None*, *cold_start='average'*, *inner_id=False*, *filter_consumed=True*, *random_rec=False*)

Recommend a list of items for given user(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids to recommend.
- **n_rec** (*int*) – Number of recommendations to return.
- **user_feats** (*None, default: None*) – NCF can't use features.
- **item_data** (*None, default: None*) – NCF can't use features.
- **cold_start** (*{'popular', 'average'}, default: 'average'*) – Cold start strategy.
  - 'popular' will sample from popular items.
  - 'average' will use the average of all the user/item embeddings as the representation of the cold-start user/item.
- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.
- **filter_consumed** (*bool, default: True*) – Whether to filter out items that a user has previously consumed.
- **random_rec** (*bool, default: False*) – Whether to choose items for recommendation based on their prediction scores.

Returns

recommendation – Recommendation result with user ids as keys and array_like recommended items as values.

Return type

dict

fit(*train_data*, *verbose=1*, *shuffle=True*, *eval_data=None*, *metrics=None*, *k=10*, *eval_batch_size=8192*, *eval_user_num=None*)

Fit TF model on the training data.

Parameters

- **train_data** (*TransformedSet* object) – Data object used for training.
- **verbose** (*int, default: 1*) – Print verbosity. If *eval_data* is provided, setting it to higher than 1 will print evaluation metrics during training.
- **shuffle** (*bool, default: True*) – Whether to shuffle the training data.
- **eval_data** (*TransformedSet* object, default: None) – Data object used for evaluating.
- **metrics** (*list or None, default: None*) – List of metrics for evaluating.
- **k** (*int, default: 10*) – Parameter of metrics, e.g. recall at k, ndcg at k
- **eval_batch_size** (*int, default: 8192*) – Batch size for evaluating.
- **eval_user_num** (*int or None, default: None*) – Number of users for evaluating. Setting it to a positive number will sample users randomly from eval data.

Raises

RuntimeError – If *fit()* is called from a loaded model(*load()*).

classmethod load(*path*, *model_name*, *data_info*, *manual=True*)

Load saved TF model for inference.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.
- **data_info** (*DataInfo* object) – Object that contains some useful information.
- **manual** (*bool, default: True*) – Whether to load model variables using numpy. If you save the model using *manual*, you should also load the mode using *manual*.

Returns

model – Loaded TF model.

Return type

type(cls)

See also:

save

rebuild_model(*path*, *model_name*, *full_assign=True*)

Assign the saved model variables to the newly initialized model.

This method is used before retraining the new model, in order to avoid training from scratch every time we get some new data.

Parameters

- **path** (*str*) – File folder path for the saved model variables.
- **model_name** (*str*) – Name of the saved model file.
- **full_assign** (*bool, default: True*) – Whether to also restore the variables of Adam optimizer.

save(*path*, *model_name*, *manual=True*, *inference_only=False*)

Save TF model for inference or retraining.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.
- **manual** (*bool, default: True*) – Whether to save model variables using numpy.
- **inference_only** (*bool, default: False*) – Whether to save model variables only for inference.

See also:

load

1.14.8 BPR

class libreco.algorithms.**BPR**(*task='ranking'*, *data_info=None*, *loss_type='bpr'*, *embed_size=16*, *n_epochs=20*, *lr=0.001*, *lr_decay=False*, *epsilon=1e-05*, *reg=None*, *batch_size=256*, *num_neg=1*, *use_tf=True*, *seed=42*, *lower_upper_bound=None*, *tf_sess_config=None*, *optimizer='adam'*, *num_threads=1*)

Bases: *EmbedBase*

Bayesian Personalized Ranking algorithm.

BPR is implemented in both TensorFlow and Cython.

Caution:

- BPR can only be used in `ranking` task.
- BPR can only use `bpr` loss in `loss_type`.

Parameters

- **task** (*{'ranking'}*) – Recommendation task. See *Task*.
- **data_info** (*DataInfo* object) – Object that contains useful information for training and inference.
- **loss_type** (*{'bpr'}*) – Loss for model training.
- **embed_size** (*int, default: 16*) – Vector size of embeddings.
- **n_epochs** (*int, default: 10*) – Number of epochs for training.
- **lr** (*float, default 0.001*) – Learning rate for training.
- **lr_decay** (*bool, default: False*) – Whether to use learning rate decay.
- **epsilon** (*float, default: 1e-5*) – A small constant added to the denominator to improve numerical stability in Adam optimizer. According to the official comment, default value of *1e-8* for *epsilon* is generally not good, so here we choose *1e-5*. Users can try tuning this hyperparameter if the training is unstable.
- **reg** (*float or None, default: None*) – Regularization parameter, must be non-negative or None.

- **batch_size** (*int, default: 256*) – Batch size for training.
- **num_neg** (*int, default: 1*) – Number of negative samples for each positive sample, only used in *ranking* task.
- **use_tf** (*bool, default: True*) – Whether to use TensorFlow or Cython version. The TensorFlow version is more accurate, whereas the Cython version is faster.
- **seed** (*int, default: 42*) – Random seed.
- **lower_upper_bound** (*tuple or None, default: None*) – Lower and upper score bound for *rating* task.
- **tf_sess_config** (*dict or None, default: None*) – Optional TensorFlow session config, see ConfigProto options.
- **optimizer** (*{'sgd', 'momentum', 'adam'}, default: 'adam'*) – Optimizer used in Cython version.
- **num_threads** (*int, default: 1*) – Number of threads used in Cython version.

References

Steffen Rendle et al. BPR: Bayesian Personalized Ranking from Implicit Feedback.

fit(*train_data*, *verbose=1*, *shuffle=True*, *eval_data=None*, *metrics=None*, *k=10*, *eval_batch_size=8192*, *eval_user_num=None*)

Fit BPR model on the training data.

Parameters

- **train_data** (*TransformedSet* object) – Data object used for training.
- **verbose** (*int, default: 1*) – Print verbosity. If *eval_data* is provided, setting it to higher than 1 will print evaluation metrics during training.
- **shuffle** (*bool, default: True*) – Whether to shuffle the training data.
- **eval_data** (*TransformedSet* object, default: None) – Data object used for evaluating.
- **metrics** (*list or None, default: None*) – List of metrics for evaluating.
- **k** (*int, default: 10*) – Parameter of metrics, e.g. recall at k, ndcg at k
- **eval_batch_size** (*int, default: 8192*) – Batch size for evaluating.
- **eval_user_num** (*int or None, default: None*) – Number of users for evaluating. Setting it to a positive number will sample users randomly from eval data.

get_item_embedding(*item=None*)

Get item embedding(s) from the model.

Parameters

item (*int or str or None*) – Query item id. If it is None, all item embeddings will be returned.

Returns

item_embedding – Returned item embeddings.

Return type

numpy.ndarray

Raises

- **ValueError** – If the item does not appear in the training data.
- **AssertionError** – If the model has not been trained.

get_user_embedding(*user=None*)

Get user embedding(s) from the model.

Parameters

user (*int or str or None*) – Query user id. If it is None, all user embeddings will be returned.

Returns

user_embedding – Returned user embeddings.

Return type

numpy.ndarray

Raises

- **ValueError** – If the user does not appear in the training data.
- **AssertionError** – If the model has not been trained.

init_knn(*approximate*, *sim_type*, *M=100*, *ef_construction=200*, *ef_search=200*)

Initialize k-nearest-search model.

Parameters

- **approximate** (*bool*) – Whether to use approximate nearest neighbor search. If it is True, nmslib must be installed. The *HNSW* method in *nmslib* is used.
- **sim_type** (*{'cosine', 'inner-product'}*) – Similarity space type.
- **M** (*int, default: 100*) – Parameter in *HNSW*, refer to nmslib doc.
- **ef_construction** (*int, default: 200*) – Parameter in *HNSW*, refer to nmslib doc.
- **ef_search** (*int, default: 200*) – Parameter in *HNSW*, refer to nmslib doc.

Raises

- **ValueError** – If sim_type is not one of ('cosine', 'inner-product').
- **ModuleNotFoundError** – If *approximate=True* and *nmslib* is not installed.

classmethod load(*path*, *model_name*, *data_info*, ***kwargs*)

Load saved embed model for inference.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.
- **data_info** (*DataInfo* object) – Object that contains some useful information.

Returns

model – Loaded embed model.

Return type

type(cls)

See also:

save

predict(*user*, *item*, *cold_start='average'*, *inner_id=False*)

Make prediction(s) on given user(s) and item(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids.
- **item** (*int or str or array_like*) – Item id or batch of item ids.
- **cold_start** (*{'popular', 'average'}, default: 'average'*) – Cold start strategy.
  - 'popular' will sample from popular items.
  - 'average' will use the average of all the user/item embeddings as the representation of the cold-start user/item.
- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.

Returns

prediction – Predicted scores for each user-item pair.

Return type

float or numpy.ndarray

rebuild_model(*path*, *model_name*, *full_assign=True*)

Assign the saved model variables to the newly initialized model.

This method is used before retraining the new model, in order to avoid training from scratch every time we get some new data.

Parameters

- **path** (*str*) – File folder path for the saved model variables.
- **model_name** (*str*) – Name of the saved model file.
- **full_assign** (*bool, default: True*) – Whether to also restore the variables of Adam optimizer.

recommend_user(*user*, *n_rec*, *cold_start='average'*, *inner_id=False*, *filter_consumed=True*, *random_rec=False*)

Recommend a list of items for given user(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids to recommend.
- **n_rec** (*int*) – Number of recommendations to return.
- **cold_start** (*{'popular', 'average'}, default: 'average'*) – Cold start strategy.
  - 'popular' will sample from popular items.
  - 'average' will use the average of all the user/item embeddings as the representation of the cold-start user/item.
- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.
- **filter_consumed** (*bool, default: True*) – Whether to filter out items that a user has previously consumed.
- **random_rec** (*bool, default: False*) – Whether to choose items for recommendation based on their prediction scores.

Returns
recommendation – Recommendation result with user ids as keys and array_like recommended items as values.

Return type
dict of {Union[int, str, array_like] : numpy.ndarray}

`save`(*path*, *model_name*, *inference_only=False*, ***kwargs*)

Save embed model for inference or retraining.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.
- **inference_only** (*bool, default: False*) – Whether to save model only for inference. If it is True, only embeddings will be saved. Otherwise, model variables will be saved.

See also:

load

`search_knn_items`(*item*, *k*)

Search most similar k items.

Parameters

- **item** (*int or str*) – Query item id.
- **k** (*int*) – Number of similar items.

Returns
similar items – A list of k similar items.

Return type
list

`search_knn_users`(*user*, *k*)

Search most similar k users.

Parameters

- **user** (*int or str*) – Query user id.
- **k** (*int*) – Number of similar users.

Returns
similar users – A list of k similar users.

Return type
list

1.14.9 Wide & Deep

class libreco.algorithms.**WideDeep**(*task*, *data_info=None*, *loss_type='cross_entropy'*, *embed_size=16*, *n_epochs=20*, *lr=None*, *lr_decay=False*, *epsilon=1e-05*, *reg=None*, *batch_size=256*, *num_neg=1*, *use_bn=True*, *dropout_rate=None*, *hidden_units=(128, 64, 32)*, *multi_sparse_combiner='sqrtn'*, *seed=42*, *lower_upper_bound=None*, *tf_sess_config=None*)

Bases: *TfBase*

Wide & Deep algorithm.

Parameters

- **task** (*{'rating', 'ranking'}*) – Recommendation task. See *Task*.
- **data_info** (*DataInfo* object) – Object that contains useful information for training and inference.
- **loss_type** (*{'cross_entropy', 'focal'}, default: 'cross_entropy'*) – Loss for model training.
- **embed_size** (*int, default: 16*) – Vector size of embeddings.
- **n_epochs** (*int, default: 10*) – Number of epochs for training.
- **lr** (*dict, default: {"wide": 0.01, "deep": 1e-4}*) – Learning rate for training. The parameter should be a dict that contains learning rate of wide and deep parts.
- **lr_decay** (*bool, default: False*) – Whether to use learning rate decay.
- **epsilon** (*float, default: 1e-5*) – A small constant added to the denominator to improve numerical stability in Adam optimizer. According to the official comment, default value of *1e-8* for *epsilon* is generally not good, so here we choose *1e-5*. Users can try tuning this hyperparameter if the training is unstable.
- **reg** (*float or None, default: None*) – Regularization parameter, must be non-negative or None.
- **batch_size** (*int, default: 256*) – Batch size for training.
- **num_neg** (*int, default: 1*) – Number of negative samples for each positive sample, only used in *ranking* task.
- **use_bn** (*bool, default: True*) – Whether to use batch normalization.
- **dropout_rate** (*float or None, default: None*) – Probability of an element to be zeroed. If it is None, dropout is not used.
- **hidden_units** (*int, list of int or tuple of (int,), default: (128, 64, 32)*) – Number of layers and corresponding layer size in MLP.

  Changed in version 1.0.0: Accept type of `int`, `list` or `tuple`, instead of `str`.
- **multi_sparse_combiner** (*{'normal', 'mean', 'sum', 'sqrtn'}, default: 'sqrtn'*) – Options for combining *multi_sparse* features.
- **seed** (*int, default: 42*) – Random seed.
- **lower_upper_bound** (*tuple or None, default: None*) – Lower and upper score bound for *rating* task.
- **tf_sess_config** (*dict or None, default: None*) – Optional TensorFlow session config, see ConfigProto options.

Notes

According to the original paper, the Wide part uses FTRL with L1 regularization as the optimizer, so we'll also adopt it here. Note this may not be suitable for your specific task.

References

Heng-Tze Cheng et al. Wide & Deep Learning for Recommender Systems.

fit(*train_data*, *verbose=1*, *shuffle=True*, *eval_data=None*, *metrics=None*, *k=10*, *eval_batch_size=8192*, *eval_user_num=None*)

Fit TF model on the training data.

Parameters

- **train_data** (*TransformedSet* object) – Data object used for training.
- **verbose** (*int, default: 1*) – Print verbosity. If *eval_data* is provided, setting it to higher than 1 will print evaluation metrics during training.
- **shuffle** (*bool, default: True*) – Whether to shuffle the training data.
- **eval_data** (*TransformedSet* object, default: None) – Data object used for evaluating.
- **metrics** (*list or None, default: None*) – List of metrics for evaluating.
- **k** (*int, default: 10*) – Parameter of metrics, e.g. recall at k, ndcg at k
- **eval_batch_size** (*int, default: 8192*) – Batch size for evaluating.
- **eval_user_num** (*int or None, default: None*) – Number of users for evaluating. Setting it to a positive number will sample users randomly from eval data.

Raises

RuntimeError – If *fit()* is called from a loaded model(*load()*).

classmethod load(*path*, *model_name*, *data_info*, *manual=True*)

Load saved TF model for inference.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.
- **data_info** (*DataInfo* object) – Object that contains some useful information.
- **manual** (*bool, default: True*) – Whether to load model variables using numpy. If you save the model using *manual*, you should also load the mode using *manual*.

Returns

model – Loaded TF model.

Return type

type(cls)

See also:

save

predict(*user*, *item*, *feats=None*, *cold_start='average'*, *inner_id=False*)

Make prediction(s) on given user(s) and item(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids.
- **item** (*int or str or array_like*) – Item id or batch of item ids.
- **feats** (*dict or pandas.Series or None, default: None*) – Extra features used in prediction.
- **cold_start** (*{'popular', 'average'}, default: 'average'*) – Cold start strategy.
  - 'popular' will sample from popular items.
  - 'average' will use the average of all the user/item embeddings as the representation of the cold-start user/item.
- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.

Returns

prediction – Predicted scores for each user-item pair.

Return type

float or numpy.ndarray

rebuild_model(*path*, *model_name*, *full_assign=True*)

Assign the saved model variables to the newly initialized model.

This method is used before retraining the new model, in order to avoid training from scratch every time we get some new data.

Parameters

- **path** (*str*) – File folder path for the saved model variables.
- **model_name** (*str*) – Name of the saved model file.
- **full_assign** (*bool, default: True*) – Whether to also restore the variables of Adam optimizer.

recommend_user(*user*, *n_rec*, *user_feats=None*, *item_data=None*, *cold_start='average'*, *inner_id=False*, *filter_consumed=True*, *random_rec=False*)

Recommend a list of items for given user(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids to recommend.
- **n_rec** (*int*) – Number of recommendations to return.
- **user_feats** (*dict or pandas.Series or None, default: None*) – Extra user features for recommendation.
- **item_data** (*pandas.DataFrame or None, default: None*) – Extra item features for recommendation.
- **cold_start** (*{'popular', 'average'}, default: 'average'*) – Cold start strategy.
  - 'popular' will sample from popular items.
  - 'average' will use the average of all the user/item embeddings as the representation of the cold-start user/item.

- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.
- **filter_consumed** (*bool, default: True*) – Whether to filter out items that a user has previously consumed.
- **random_rec** (*bool, default: False*) – Whether to choose items for recommendation based on their prediction scores.

Returns

recommendation – Recommendation result with user ids as keys and array_like recommended items as values.

Return type

dict of {Union[int, str, array_like] : numpy.ndarray}

save(*path*, *model_name*, *manual=True*, *inference_only=False*)

Save TF model for inference or retraining.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.
- **manual** (*bool, default: True*) – Whether to save model variables using numpy.
- **inference_only** (*bool, default: False*) – Whether to save model variables only for inference.

See also:

load

1.14.10 FM

class libreco.algorithms.**FM**(*task*, *data_info*, *loss_type='cross_entropy'*, *embed_size=16*, *n_epochs=20*, *lr=0.001*, *lr_decay=False*, *epsilon=1e-05*, *reg=None*, *batch_size=256*, *num_neg=1*, *use_bn=True*, *dropout_rate=None*, *multi_sparse_combiner='sqrtn'*, *seed=42*, *lower_upper_bound=None*, *tf_sess_config=None*)

Bases: *TfBase*

Factorization Machines algorithm.

Note this implementation is actually a mixture of FM and NFM, since it uses one dense layer in the final output

Parameters

- **task** (*{'rating', 'ranking'}*) – Recommendation task. See *Task*.
- **data_info** (*DataInfo* object) – Object that contains useful information for training and inference.
- **loss_type** (*{'cross_entropy', 'focal'}, default: 'cross_entropy'*) – Loss for model training.
- **embed_size** (*int, default: 16*) – Vector size of embeddings.
- **n_epochs** (*int, default: 10*) – Number of epochs for training.
- **lr** (*float, default 0.001*) – Learning rate for training.

- **lr_decay** (*bool, default: False*) – Whether to use learning rate decay.
- **epsilon** (*float, default: 1e-5*) – A small constant added to the denominator to improve numerical stability in Adam optimizer. According to the official comment, default value of *1e-8* for *epsilon* is generally not good, so here we choose *1e-5*. Users can try tuning this hyperparameter if the training is unstable.
- **reg** (*float or None, default: None*) – Regularization parameter, must be non-negative or None.
- **batch_size** (*int, default: 256*) – Batch size for training.
- **num_neg** (*int, default: 1*) – Number of negative samples for each positive sample, only used in *ranking* task.
- **use_bn** (*bool, default: True*) – Whether to use batch normalization.
- **dropout_rate** (*float or None, default: None*) – Probability of an element to be zeroed. If it is None, dropout is not used.
- **multi_sparse_combiner** (*{'normal', 'mean', 'sum', 'sqrtn'}, default: 'sqrtn'*) – Options for combining *multi_sparse* features.
- **seed** (*int, default: 42*) – Random seed.
- **lower_upper_bound** (*tuple or None, default: None*) – Lower and upper score bound for *rating* task.
- **tf_sess_config** (*dict or None, default: None*) – Optional TensorFlow session config, see ConfigProto options.

References

[1] *Steffen Rendle* Factorization Machines.

[2] *Xiangnan He et al.* Neural Factorization Machines for Sparse Predictive Analytics.

fit(*train_data*, *verbose=1*, *shuffle=True*, *eval_data=None*, *metrics=None*, *k=10*, *eval_batch_size=8192*, *eval_user_num=None*)

Fit TF model on the training data.

Parameters

- **train_data** (*TransformedSet* object) – Data object used for training.
- **verbose** (*int, default: 1*) – Print verbosity. If *eval_data* is provided, setting it to higher than 1 will print evaluation metrics during training.
- **shuffle** (*bool, default: True*) – Whether to shuffle the training data.
- **eval_data** (*TransformedSet* object, default: None) – Data object used for evaluating.
- **metrics** (*list or None, default: None*) – List of metrics for evaluating.
- **k** (*int, default: 10*) – Parameter of metrics, e.g. recall at k, ndcg at k
- **eval_batch_size** (*int, default: 8192*) – Batch size for evaluating.
- **eval_user_num** (*int or None, default: None*) – Number of users for evaluating. Setting it to a positive number will sample users randomly from eval data.

Raises

RuntimeError – If *fit()* is called from a loaded model(*load()*).

classmethod load(*path*, *model_name*, *data_info*, *manual=True*)

Load saved TF model for inference.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.
- **data_info** (*DataInfo* object) – Object that contains some useful information.
- **manual** (*bool, default: True*) – Whether to load model variables using numpy. If you save the model using *manual*, you should also load the mode using *manual*.

Returns

model – Loaded TF model.

Return type

type(cls)

See also:

save

predict(*user*, *item*, *feats=None*, *cold_start='average'*, *inner_id=False*)

Make prediction(s) on given user(s) and item(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids.
- **item** (*int or str or array_like*) – Item id or batch of item ids.
- **feats** (*dict or pandas.Series or None, default: None*) – Extra features used in prediction.
- **cold_start** (*{'popular', 'average'}, default: 'average'*) – Cold start strategy.
  - 'popular' will sample from popular items.
  - 'average' will use the average of all the user/item embeddings as the representation of the cold-start user/item.
- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.

Returns

prediction – Predicted scores for each user-item pair.

Return type

float or numpy.ndarray

rebuild_model(*path*, *model_name*, *full_assign=True*)

Assign the saved model variables to the newly initialized model.

This method is used before retraining the new model, in order to avoid training from scratch every time we get some new data.

Parameters

- **path** (*str*) – File folder path for the saved model variables.
- **model_name** (*str*) – Name of the saved model file.
- **full_assign** (*bool, default: True*) – Whether to also restore the variables of Adam optimizer.

recommend_user(*user*, *n_rec*, *user_feats=None*, *item_data=None*, *cold_start='average'*, *inner_id=False*, *filter_consumed=True*, *random_rec=False*)

Recommend a list of items for given user(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids to recommend.
- **n_rec** (*int*) – Number of recommendations to return.
- **user_feats** (*dict or pandas.Series or None, default: None*) – Extra user features for recommendation.
- **item_data** (*pandas.DataFrame or None, default: None*) – Extra item features for recommendation.
- **cold_start** (*{'popular', 'average'}, default: 'average'*) – Cold start strategy.
  - 'popular' will sample from popular items.
  - 'average' will use the average of all the user/item embeddings as the representation of the cold-start user/item.
- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.
- **filter_consumed** (*bool, default: True*) – Whether to filter out items that a user has previously consumed.
- **random_rec** (*bool, default: False*) – Whether to choose items for recommendation based on their prediction scores.

Returns

recommendation – Recommendation result with user ids as keys and array_like recommended items as values.

Return type

dict of {Union[int, str, array_like] : numpy.ndarray}

save(*path*, *model_name*, *manual=True*, *inference_only=False*)

Save TF model for inference or retraining.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.
- **manual** (*bool, default: True*) – Whether to save model variables using numpy.
- **inference_only** (*bool, default: False*) – Whether to save model variables only for inference.

See also:

load

1.14.11 DeepFM

class libreco.algorithms.**DeepFM**(*task*, *data_info*, *loss_type='cross_entropy'*, *embed_size=16*, *n_epochs=20*, *lr=0.001*, *lr_decay=False*, *epsilon=1e-05*, *reg=None*, *batch_size=256*, *num_neg=1*, *use_bn=True*, *dropout_rate=None*, *hidden_units=(128, 64, 32)*, *multi_sparse_combiner='sqrtn'*, *seed=42*, *lower_upper_bound=None*, *tf_sess_config=None*)

Bases: *TfBase*

DeepFM algorithm.

Parameters

- **task** (*{'rating', 'ranking'}*) – Recommendation task. See *Task*.
- **data_info** (*DataInfo* object) – Object that contains useful information for training and inference.
- **loss_type** (*{'cross_entropy', 'focal'}, default: 'cross_entropy'*) – Loss for model training.
- **embed_size** (*int, default: 16*) – Vector size of embeddings.
- **n_epochs** (*int, default: 10*) – Number of epochs for training.
- **lr** (*float, default 0.001*) – Learning rate for training.
- **lr_decay** (*bool, default: False*) – Whether to use learning rate decay.
- **epsilon** (*float, default: 1e-5*) – A small constant added to the denominator to improve numerical stability in Adam optimizer. According to the official comment, default value of *1e-8* for *epsilon* is generally not good, so here we choose *1e-5*. Users can try tuning this hyperparameter if the training is unstable.
- **reg** (*float or None, default: None*) – Regularization parameter, must be non-negative or None.
- **batch_size** (*int, default: 256*) – Batch size for training.
- **num_neg** (*int, default: 1*) – Number of negative samples for each positive sample, only used in *ranking* task.
- **use_bn** (*bool, default: True*) – Whether to use batch normalization.
- **dropout_rate** (*float or None, default: None*) – Probability of an element to be zeroed. If it is None, dropout is not used.
- **hidden_units** (*int, list of int or tuple of (int,), default: (128, 64, 32)*) – Number of layers and corresponding layer size in MLP.

  Changed in version 1.0.0: Accept type of `int`, `list` or `tuple`, instead of `str`.
- **multi_sparse_combiner** (*{'normal', 'mean', 'sum', 'sqrtn'}, default: 'sqrtn'*) – Options for combining *multi_sparse* features.
- **seed** (*int, default: 42*) – Random seed.
- **lower_upper_bound** (*tuple or None, default: None*) – Lower and upper score bound for *rating* task.
- **tf_sess_config** (*dict or None, default: None*) – Optional TensorFlow session config, see ConfigProto options.

References

Huifeng Guo et al. DeepFM: A Factorization-Machine based Neural Network for CTR Prediction.

fit(*train_data*, *verbose=1*, *shuffle=True*, *eval_data=None*, *metrics=None*, *k=10*, *eval_batch_size=8192*, *eval_user_num=None*)

Fit TF model on the training data.

Parameters

- **train_data** (*TransformedSet* object) – Data object used for training.
- **verbose** (*int, default: 1*) – Print verbosity. If *eval_data* is provided, setting it to higher than 1 will print evaluation metrics during training.
- **shuffle** (*bool, default: True*) – Whether to shuffle the training data.
- **eval_data** (*TransformedSet* object, default: None) – Data object used for evaluating.
- **metrics** (*list or None, default: None*) – List of metrics for evaluating.
- **k** (*int, default: 10*) – Parameter of metrics, e.g. recall at k, ndcg at k
- **eval_batch_size** (*int, default: 8192*) – Batch size for evaluating.
- **eval_user_num** (*int or None, default: None*) – Number of users for evaluating. Setting it to a positive number will sample users randomly from eval data.

Raises

RuntimeError – If *fit()* is called from a loaded model(*load()*).

classmethod load(*path*, *model_name*, *data_info*, *manual=True*)

Load saved TF model for inference.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.
- **data_info** (*DataInfo* object) – Object that contains some useful information.
- **manual** (*bool, default: True*) – Whether to load model variables using numpy. If you save the model using *manual*, you should also load the mode using *manual*.

Returns

model – Loaded TF model.

Return type

type(cls)

See also:

save

predict(*user*, *item*, *feats=None*, *cold_start='average'*, *inner_id=False*)

Make prediction(s) on given user(s) and item(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids.
- **item** (*int or str or array_like*) – Item id or batch of item ids.

- **feats** (*dict or pandas.Series or None, default: None*) – Extra features used in prediction.
- **cold_start** (*{'popular', 'average'}, default: 'average'*) – Cold start strategy.
  - 'popular' will sample from popular items.
  - 'average' will use the average of all the user/item embeddings as the representation of the cold-start user/item.
- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.

Returns
prediction – Predicted scores for each user-item pair.

Return type
float or numpy.ndarray

rebuild_model(*path*, *model_name*, *full_assign=True*)

Assign the saved model variables to the newly initialized model.

This method is used before retraining the new model, in order to avoid training from scratch every time we get some new data.

Parameters

- **path** (*str*) – File folder path for the saved model variables.
- **model_name** (*str*) – Name of the saved model file.
- **full_assign** (*bool, default: True*) – Whether to also restore the variables of Adam optimizer.

recommend_user(*user*, *n_rec*, *user_feats=None*, *item_data=None*, *cold_start='average'*, *inner_id=False*, *filter_consumed=True*, *random_rec=False*)

Recommend a list of items for given user(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids to recommend.
- **n_rec** (*int*) – Number of recommendations to return.
- **user_feats** (*dict or pandas.Series or None, default: None*) – Extra user features for recommendation.
- **item_data** (*pandas.DataFrame or None, default: None*) – Extra item features for recommendation.
- **cold_start** (*{'popular', 'average'}, default: 'average'*) – Cold start strategy.
  - 'popular' will sample from popular items.
  - 'average' will use the average of all the user/item embeddings as the representation of the cold-start user/item.
- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.
- **filter_consumed** (*bool, default: True*) – Whether to filter out items that a user has previously consumed.
- **random_rec** (*bool, default: False*) – Whether to choose items for recommendation based on their prediction scores.

Returns

recommendation – Recommendation result with user ids as keys and array_like recommended items as values.

Return type

dict of {Union[int, str, array_like] : numpy.ndarray}

save(*path*, *model_name*, *manual=True*, *inference_only=False*)

Save TF model for inference or retraining.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.
- **manual** (*bool, default: True*) – Whether to save model variables using numpy.
- **inference_only** (*bool, default: False*) – Whether to save model variables only for inference.

See also:

load

1.14.12 YouTubeRetrieval

class libreco.algorithms.**YouTubeRetrieval**(*task='ranking'*, *data_info=None*, *loss_type='sampled_softmax'*, *embed_size=16*, *n_epochs=20*, *lr=0.001*, *lr_decay=False*, *epsilon=1e-05*, *reg=None*, *batch_size=256*, *use_bn=True*, *dropout_rate=None*, *hidden_units=(128, 64)*, *num_sampled_per_batch=None*, *sampler='uniform'*, *recent_num=10*, *random_num=None*, *multi_sparse_combiner='sqrtn'*, *seed=42*, *lower_upper_bound=None*, *tf_sess_config=None*)

Bases: *EmbedBase*

YouTubeRetrieval algorithm.

Note: The algorithm implemented mainly corresponds to the candidate generation phase based on the original paper.

Warning: YouTubeRetrieval can only be used in *ranking* task.

Parameters

- **task** (*{'ranking'}*) – Recommendation task. See *Task*.
- **data_info** (*DataInfo* object) – Object that contains useful information for training and inference.
- **loss_type** (*{'sampled_softmax', 'nce'}, default: 'sampled_softmax'*) – Loss for model training.
- **embed_size** (*int, default: 16*) – Vector size of embeddings.
- **n_epochs** (*int, default: 10*) – Number of epochs for training.

- **lr** (*float, default 0.001*) – Learning rate for training.
- **lr_decay** (*bool, default: False*) – Whether to use learning rate decay.
- **epsilon** (*float, default: 1e-5*) – A small constant added to the denominator to improve numerical stability in Adam optimizer. According to the official comment, default value of *1e-8* for *epsilon* is generally not good, so here we choose *1e-5*. Users can try tuning this hyperparameter if the training is unstable.
- **reg** (*float or None, default: None*) – Regularization parameter, must be non-negative or None.
- **batch_size** (*int, default: 256*) – Batch size for training.
- **use_bn** (*bool, default: True*) – Whether to use batch normalization.
- **dropout_rate** (*float or None, default: None*) – Probability of an element to be zeroed. If it is None, dropout is not used.
- **hidden_units** (*int, list of int or tuple of (int,), default: (128, 64)*) – Number of layers and corresponding layer size in MLP.

  Changed in version 1.0.0: Accept type of `int`, `list` or `tuple`, instead of `str`.
- **num_sampled_per_batch** (*int or None, default: None*) – Number of negative samples in a batch. If None, it is set to *batch_size*.
- **sampler** (*str, default: 'uniform'*) – Negative Sampling strategy. 'uniform' will use uniform sampler, and setting to other value will use *log_uniform_candidate_sampler* in TensorFlow. In recommendation scenarios the uniform sampler is generally preferred.
- **recent_num** (*int or None, default: 10*) – Number of recent items to use in user behavior sequence.
- **random_num** (*int or None, default: None*) – Number of random sampled items to use in user behavior sequence. If *recent_num* is not None, *random_num* is not considered.
- **multi_sparse_combiner** (*{'normal', 'mean', 'sum', 'sqrtn'}, default: 'sqrtn'*) – Options for combining *multi_sparse* features.
- **seed** (*int, default: 42*) – Random seed.
- **lower_upper_bound** (*tuple or None, default: None*) – Lower and upper score bound for *rating* task.
- **tf_sess_config** (*dict or None, default: None*) – Optional TensorFlow session config, see ConfigProto options.

References

Paul Covington et al. Deep Neural Networks for YouTube Recommendations.

fit(*train_data*, *verbose=1*, *shuffle=True*, *eval_data=None*, *metrics=None*, *k=10*, *eval_batch_size=8192*, *eval_user_num=None*)

Fit embed model on the training data.

Parameters

- **train_data** (*TransformedSet* object) – Data object used for training.
- **verbose** (*int, default: 1*) – Print verbosity. If *eval_data* is provided, setting it to higher than 1 will print evaluation metrics during training.

- **shuffle** (*bool, default: True*) – Whether to shuffle the training data.
- **eval_data** (*TransformedSet* object, default: None) – Data object used for evaluating.
- **metrics** (*list or None, default: None*) – List of metrics for evaluating.
- **k** (*int, default: 10*) – Parameter of metrics, e.g. recall at k, ndcg at k
- **eval_batch_size** (*int, default: 8192*) – Batch size for evaluating.
- **eval_user_num** (*int or None, default: None*) – Number of users for evaluating. Setting it to a positive number will sample users randomly from eval data.

Raises
RuntimeError – If *fit()* is called from a loaded model(*load()*).

get_item_embedding(*item=None*)

Get item embedding(s) from the model.

Parameters
item (*int or str or None*) – Query item id. If it is None, all item embeddings will be returned.

Returns
item_embedding – Returned item embeddings.

Return type
numpy.ndarray

Raises
- **ValueError** – If the item does not appear in the training data.
- **AssertionError** – If the model has not been trained.

get_user_embedding(*user=None*)

Get user embedding(s) from the model.

Parameters
user (*int or str or None*) – Query user id. If it is None, all user embeddings will be returned.

Returns
user_embedding – Returned user embeddings.

Return type
numpy.ndarray

Raises
- **ValueError** – If the user does not appear in the training data.
- **AssertionError** – If the model has not been trained.

init_knn(*approximate*, *sim_type*, *M=100*, *ef_construction=200*, *ef_search=200*)

Initialize k-nearest-search model.

Parameters
- **approximate** (*bool*) – Whether to use approximate nearest neighbor search. If it is True, nmslib must be installed. The *HNSW* method in *nmslib* is used.
- **sim_type** (*{'cosine', 'inner-product'}*) – Similarity space type.
- **M** (*int, default: 100*) – Parameter in *HNSW*, refer to nmslib doc.

- **ef_construction** (*int, default: 200*) – Parameter in *HNSW*, refer to nmslib doc.
- **ef_search** (*int, default: 200*) – Parameter in *HNSW*, refer to nmslib doc.

Raises

- **ValueError** – If sim_type is not one of ('cosine', 'inner-product').
- **ModuleNotFoundError** – If *approximate=True* and *nmslib* is not installed.

classmethod load(*path*, *model_name*, *data_info*, ***kwargs*)

Load saved embed model for inference.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.
- **data_info** (*DataInfo* object) – Object that contains some useful information.

Returns
model – Loaded embed model.

Return type
type(cls)

See also:

save

predict(*user*, *item*, *cold_start='average'*, *inner_id=False*)

Make prediction(s) on given user(s) and item(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids.
- **item** (*int or str or array_like*) – Item id or batch of item ids.
- **cold_start** (*{'popular', 'average'}, default: 'average'*) – Cold start strategy.
  - 'popular' will sample from popular items.
  - 'average' will use the average of all the user/item embeddings as the representation of the cold-start user/item.
- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.

Returns
prediction – Predicted scores for each user-item pair.

Return type
float or numpy.ndarray

rebuild_model(*path*, *model_name*, *full_assign=True*)

Assign the saved model variables to the newly initialized model.

This method is used before retraining the new model, in order to avoid training from scratch every time we get some new data.

Parameters

- **path** (*str*) – File folder path for the saved model variables.
- **model_name** (*str*) – Name of the saved model file.

- **full_assign** (*bool, default: True*) – Whether to also restore the variables of Adam optimizer.

recommend_user(*user*, *n_rec*, *cold_start='average'*, *inner_id=False*, *filter_consumed=True*, *random_rec=False*)

Recommend a list of items for given user(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids to recommend.
- **n_rec** (*int*) – Number of recommendations to return.
- **cold_start** (*{'popular', 'average'}, default: 'average'*) – Cold start strategy.
  - 'popular' will sample from popular items.
  - 'average' will use the average of all the user/item embeddings as the representation of the cold-start user/item.
- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.
- **filter_consumed** (*bool, default: True*) – Whether to filter out items that a user has previously consumed.
- **random_rec** (*bool, default: False*) – Whether to choose items for recommendation based on their prediction scores.

Returns

recommendation – Recommendation result with user ids as keys and array_like recommended items as values.

Return type

dict of {Union[int, str, array_like] : numpy.ndarray}

save(*path*, *model_name*, *inference_only=False*, ***kwargs*)

Save embed model for inference or retraining.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.
- **inference_only** (*bool, default: False*) – Whether to save model only for inference. If it is True, only embeddings will be saved. Otherwise, model variables will be saved.

See also:

load

search_knn_items(*item*, *k*)

Search most similar k items.

Parameters

- **item** (*int or str*) – Query item id.
- **k** (*int*) – Number of similar items.

Returns

similar items – A list of k similar items.

Return type
list

`search_knn_users`(*user*, *k*)

Search most similar k users.

Parameters

- **user** (*int or str*) – Query user id.
- **k** (*int*) – Number of similar users.

Returns
similar users – A list of k similar users.

Return type
list

1.14.13 YouTubeRanking

`class libreco.algorithms.YouTubeRanking`(*task='ranking'*, *data_info=None*, *loss_type='cross_entropy'*, *embed_size=16*, *n_epochs=20*, *lr=0.001*, *lr_decay=False*, *epsilon=1e-05*, *reg=None*, *batch_size=256*, *num_neg=1*, *use_bn=True*, *dropout_rate=None*, *hidden_units=(128, 64, 32)*, *recent_num=10*, *random_num=None*, *multi_sparse_combiner='sqrtn'*, *seed=42*, *lower_upper_bound=None*, *tf_sess_config=None*)

Bases: *TfBase*

YouTubeRanking algorithm.

Note: The algorithm implemented mainly corresponds to the ranking phase based on the original paper.

Warning: YouTubeRanking can only be used in *ranking* task.

Parameters

- **task** (*{'ranking'}*) – Recommendation task. See *Task*.
- **data_info** (*DataInfo* object) – Object that contains useful information for training and inference.
- **loss_type** (*{'cross_entropy', 'focal'}, default: 'cross_entropy'*) – Loss for model training.
- **embed_size** (*int, default: 16*) – Vector size of embeddings.
- **n_epochs** (*int, default: 10*) – Number of epochs for training.
- **lr** (*float, default 0.001*) – Learning rate for training.
- **lr_decay** (*bool, default: False*) – Whether to use learning rate decay.
- **epsilon** (*float, default: 1e-5*) – A small constant added to the denominator to improve numerical stability in Adam optimizer. According to the official comment, default value of *1e-8* for *epsilon* is generally not good, so here we choose *1e-5*. Users can try tuning this hyperparameter if the training is unstable.

- **reg** (*float or None, default: None*) – Regularization parameter, must be non-negative or None.
- **batch_size** (*int, default: 256*) – Batch size for training.
- **num_neg** (*int, default: 1*) – Number of negative samples for each positive sample, only used in *ranking* task.
- **use_bn** (*bool, default: True*) – Whether to use batch normalization.
- **dropout_rate** (*float or None, default: None*) – Probability of an element to be zeroed. If it is None, dropout is not used.
- **hidden_units** (*int, list of int or tuple of (int,), default: (128, 64, 32)*) – Number of layers and corresponding layer size in MLP.

  Changed in version 1.0.0: Accept type of `int`, `list` or `tuple`, instead of `str`.
- **recent_num** (*int or None, default: 10*) – Number of recent items to use in user behavior sequence.
- **random_num** (*int or None, default: None*) – Number of random sampled items to use in user behavior sequence. If *recent_num* is not None, *random_num* is not considered.
- **multi_sparse_combiner** (*{'normal', 'mean', 'sum', 'sqrtn'}, default: 'sqrtn'*) – Options for combining *multi_sparse* features.
- **seed** (*int, default: 42*) – Random seed.
- **lower_upper_bound** (*tuple or None, default: None*) – Lower and upper score bound for *rating* task.
- **tf_sess_config** (*dict or None, default: None*) – Optional TensorFlow session config, see ConfigProto options.

References

Paul Covington et al. Deep Neural Networks for YouTube Recommendations.

fit(*train_data*, *verbose=1*, *shuffle=True*, *eval_data=None*, *metrics=None*, *k=10*, *eval_batch_size=8192*, *eval_user_num=None*)

Fit TF model on the training data.

Parameters

- **train_data** (*TransformedSet* object) – Data object used for training.
- **verbose** (*int, default: 1*) – Print verbosity. If *eval_data* is provided, setting it to higher than 1 will print evaluation metrics during training.
- **shuffle** (*bool, default: True*) – Whether to shuffle the training data.
- **eval_data** (*TransformedSet* object, default: None) – Data object used for evaluating.
- **metrics** (*list or None, default: None*) – List of metrics for evaluating.
- **k** (*int, default: 10*) – Parameter of metrics, e.g. recall at k, ndcg at k
- **eval_batch_size** (*int, default: 8192*) – Batch size for evaluating.
- **eval_user_num** (*int or None, default: None*) – Number of users for evaluating. Setting it to a positive number will sample users randomly from eval data.

Raises
RuntimeError – If *fit()* is called from a loaded model(*load()*).

classmethod load(*path*, *model_name*, *data_info*, *manual=True*)

Load saved TF model for inference.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.
- **data_info** (*DataInfo* object) – Object that contains some useful information.
- **manual** (*bool, default: True*) – Whether to load model variables using numpy. If you save the model using *manual*, you should also load the mode using *manual*.

Returns
model – Loaded TF model.

Return type
type(cls)

See also:

save

predict(*user*, *item*, *feats=None*, *cold_start='average'*, *inner_id=False*)

Make prediction(s) on given user(s) and item(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids.
- **item** (*int or str or array_like*) – Item id or batch of item ids.
- **feats** (*dict or pandas.Series or None, default: None*) – Extra features used in prediction.
- **cold_start** (*{'popular', 'average'}, default: 'average'*) – Cold start strategy.
  - 'popular' will sample from popular items.
  - 'average' will use the average of all the user/item embeddings as the representation of the cold-start user/item.
- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.

Returns
prediction – Predicted scores for each user-item pair.

Return type
float or numpy.ndarray

rebuild_model(*path*, *model_name*, *full_assign=True*)

Assign the saved model variables to the newly initialized model.

This method is used before retraining the new model, in order to avoid training from scratch every time we get some new data.

Parameters

- **path** (*str*) – File folder path for the saved model variables.
- **model_name** (*str*) – Name of the saved model file.

- **full_assign** (*bool, default: True*) – Whether to also restore the variables of Adam optimizer.

recommend_user(*user*, *n_rec*, *user_feats=None*, *item_data=None*, *cold_start='average'*, *inner_id=False*, *filter_consumed=True*, *random_rec=False*)

Recommend a list of items for given user(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids to recommend.
- **n_rec** (*int*) – Number of recommendations to return.
- **user_feats** (*dict or pandas.Series or None, default: None*) – Extra user features for recommendation.
- **item_data** (*pandas.DataFrame or None, default: None*) – Extra item features for recommendation.
- **cold_start** (*{'popular', 'average'}, default: 'average'*) – Cold start strategy.
  - 'popular' will sample from popular items.
  - 'average' will use the average of all the user/item embeddings as the representation of the cold-start user/item.
- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.
- **filter_consumed** (*bool, default: True*) – Whether to filter out items that a user has previously consumed.
- **random_rec** (*bool, default: False*) – Whether to choose items for recommendation based on their prediction scores.

Returns

recommendation – Recommendation result with user ids as keys and array_like recommended items as values.

Return type

dict of {Union[int, str, array_like] : numpy.ndarray}

save(*path*, *model_name*, *manual=True*, *inference_only=False*)

Save TF model for inference or retraining.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.
- **manual** (*bool, default: True*) – Whether to save model variables using numpy.
- **inference_only** (*bool, default: False*) – Whether to save model variables only for inference.

See also:

load

1.14.14 AutoInt

class libreco.algorithms.**AutoInt**(*task*, *data_info*, *loss_type='cross_entropy'*, *embed_size=16*, *n_epochs=10*, *lr=0.001*, *lr_decay=False*, *epsilon=1e-05*, *reg=None*, *batch_size=256*, *num_neg=1*, *use_bn=True*, *dropout_rate=None*, *att_embed_size=(8, 8, 8)*, *num_heads=2*, *use_residual=True*, *multi_sparse_combiner='sqrtn'*, *seed=42*, *lower_upper_bound=None*, *tf_sess_config=None*)

Bases: *TfBase*

AutoInt algorithm.

Parameters

- **task** (*{'rating', 'ranking'}*) – Recommendation task. See *Task*.
- **data_info** (*DataInfo* object) – Object that contains useful information for training and inference.
- **loss_type** (*{'cross_entropy', 'focal'}, default: 'cross_entropy'*) – Loss for model training.
- **embed_size** (*int, default: 16*) – Vector size of embeddings.
- **n_epochs** (*int, default: 10*) – Number of epochs for training.
- **lr** (*float, default 0.001*) – Learning rate for training.
- **lr_decay** (*bool, default: False*) – Whether to use learning rate decay.
- **epsilon** (*float, default: 1e-5*) – A small constant added to the denominator to improve numerical stability in Adam optimizer. According to the official comment, default value of *1e-8* for *epsilon* is generally not good, so here we choose *1e-5*. Users can try tuning this hyperparameter if the training is unstable.
- **reg** (*float or None, default: None*) – Regularization parameter, must be non-negative or None.
- **batch_size** (*int, default: 256*) – Batch size for training.
- **num_neg** (*int, default: 1*) – Number of negative samples for each positive sample, only used in *ranking* task.
- **use_bn** (*bool, default: True*) – Whether to use batch normalization.
- **dropout_rate** (*float or None, default: None*) – Probability of an element to be zeroed. If it is None, dropout is not used.
- **att_embed_size** (*int, list of int or tuple of (int,), default: (8, 8, 8)*) – Embedding size in each attention layer. If it is *int*, one layer is used.
- **num_heads** (*int, default: 2*) – Number of heads in multi-head attention.
- **use_residual** (*bool, default: True*) – Whether to use residual layer.
- **multi_sparse_combiner** (*{'normal', 'mean', 'sum', 'sqrtn'}, default: 'sqrtn'*) – Options for combining *multi_sparse* features.
- **seed** (*int, default: 42*) – Random seed.
- **lower_upper_bound** (*tuple or None, default: None*) – Lower and upper score bound for *rating* task.

- **tf_sess_config** (*dict or None, default: None*) – Optional TensorFlow session config, see ConfigProto options.

References

Weiping Song et al. AutoInt: Automatic Feature Interaction Learning via Self-Attentive Neural Networks.

fit(*train_data*, *verbose=1*, *shuffle=True*, *eval_data=None*, *metrics=None*, *k=10*, *eval_batch_size=8192*, *eval_user_num=None*)

Fit TF model on the training data.

Parameters

- **train_data** (*TransformedSet* object) – Data object used for training.
- **verbose** (*int, default: 1*) – Print verbosity. If *eval_data* is provided, setting it to higher than 1 will print evaluation metrics during training.
- **shuffle** (*bool, default: True*) – Whether to shuffle the training data.
- **eval_data** (*TransformedSet* object, default: None) – Data object used for evaluating.
- **metrics** (*list or None, default: None*) – List of metrics for evaluating.
- **k** (*int, default: 10*) – Parameter of metrics, e.g. recall at k, ndcg at k
- **eval_batch_size** (*int, default: 8192*) – Batch size for evaluating.
- **eval_user_num** (*int or None, default: None*) – Number of users for evaluating. Setting it to a positive number will sample users randomly from eval data.

Raises

RuntimeError – If *fit()* is called from a loaded model(*load()*).

classmethod load(*path*, *model_name*, *data_info*, *manual=True*)

Load saved TF model for inference.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.
- **data_info** (*DataInfo* object) – Object that contains some useful information.
- **manual** (*bool, default: True*) – Whether to load model variables using numpy. If you save the model using *manual*, you should also load the mode using *manual*.

Returns

model – Loaded TF model.

Return type

type(cls)

See also:

save

predict(*user*, *item*, *feats=None*, *cold_start='average'*, *inner_id=False*)

Make prediction(s) on given user(s) and item(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids.

- **item** (*int or str or array_like*) – Item id or batch of item ids.
- **feats** (*dict or pandas.Series or None, default: None*) – Extra features used in prediction.
- **cold_start** (*{'popular', 'average'}, default: 'average'*) – Cold start strategy.
  - 'popular' will sample from popular items.
  - 'average' will use the average of all the user/item embeddings as the representation of the cold-start user/item.
- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.

Returns
prediction – Predicted scores for each user-item pair.

Return type
float or numpy.ndarray

rebuild_model(*path*, *model_name*, *full_assign=True*)

Assign the saved model variables to the newly initialized model.

This method is used before retraining the new model, in order to avoid training from scratch every time we get some new data.

Parameters

- **path** (*str*) – File folder path for the saved model variables.
- **model_name** (*str*) – Name of the saved model file.
- **full_assign** (*bool, default: True*) – Whether to also restore the variables of Adam optimizer.

recommend_user(*user*, *n_rec*, *user_feats=None*, *item_data=None*, *cold_start='average'*, *inner_id=False*, *filter_consumed=True*, *random_rec=False*)

Recommend a list of items for given user(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids to recommend.
- **n_rec** (*int*) – Number of recommendations to return.
- **user_feats** (*dict or pandas.Series or None, default: None*) – Extra user features for recommendation.
- **item_data** (*pandas.DataFrame or None, default: None*) – Extra item features for recommendation.
- **cold_start** (*{'popular', 'average'}, default: 'average'*) – Cold start strategy.
  - 'popular' will sample from popular items.
  - 'average' will use the average of all the user/item embeddings as the representation of the cold-start user/item.
- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.
- **filter_consumed** (*bool, default: True*) – Whether to filter out items that a user has previously consumed.

- **random_rec** (*bool, default: False*) – Whether to choose items for recommendation based on their prediction scores.

Returns

recommendation – Recommendation result with user ids as keys and array_like recommended items as values.

Return type

dict of {Union[int, str, array_like] : numpy.ndarray}

save(*path*, *model_name*, *manual=True*, *inference_only=False*)

Save TF model for inference or retraining.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.
- **manual** (*bool, default: True*) – Whether to save model variables using numpy.
- **inference_only** (*bool, default: False*) – Whether to save model variables only for inference.

See also:

load

1.14.15 DIN

class libreco.algorithms.**DIN**(*task*, *data_info=None*, *loss_type='cross_entropy'*, *embed_size=16*, *n_epochs=20*, *lr=0.001*, *lr_decay=False*, *epsilon=1e-05*, *reg=None*, *batch_size=256*, *num_neg=1*, *use_bn=True*, *dropout_rate=None*, *hidden_units=(128, 64, 32)*, *recent_num=10*, *random_num=None*, *use_tf_attention=False*, *multi_sparse_combiner='sqrtn'*, *seed=42*, *lower_upper_bound=None*, *tf_sess_config=None*)

Bases: *TfBase*

Deep Interest Network algorithm.

Parameters

- **task** (*{'rating', 'ranking'}*) – Recommendation task. See *Task*.
- **data_info** (*DataInfo* object) – Object that contains useful information for training and inference.
- **loss_type** (*{'cross_entropy', 'focal'}, default: 'cross_entropy'*) – Loss for model training.
- **embed_size** (*int, default: 16*) – Vector size of embeddings.
- **n_epochs** (*int, default: 10*) – Number of epochs for training.
- **lr** (*float, default 0.001*) – Learning rate for training.
- **lr_decay** (*bool, default: False*) – Whether to use learning rate decay.
- **epsilon** (*float, default: 1e-5*) – A small constant added to the denominator to improve numerical stability in Adam optimizer. According to the official comment, default value of *1e-8* for *epsilon* is generally not good, so here we choose *1e-5*. Users can try tuning this hyperparameter if the training is unstable.

- **reg** (*float or None, default: None*) – Regularization parameter, must be non-negative or None.
- **batch_size** (*int, default: 256*) – Batch size for training.
- **num_neg** (*int, default: 1*) – Number of negative samples for each positive sample, only used in *ranking* task.
- **use_bn** (*bool, default: True*) – Whether to use batch normalization.
- **dropout_rate** (*float or None, default: None*) – Probability of an element to be zeroed. If it is None, dropout is not used.
- **hidden_units** (*int, list of int or tuple of (int,), default: (128, 64, 32)*) – Number of layers and corresponding layer size in MLP.

  Changed in version 1.0.0: Accept type of `int`, `list` or `tuple`, instead of `str`.
- **recent_num** (*int or None, default: 10*) – Number of recent items to use in user behavior sequence.
- **random_num** (*int or None, default: None*) – Number of random sampled items to use in user behavior sequence. If *recent_num* is not None, *random_num* is not considered.
- **use_tf_attention** (*bool, default: False*) – Whether to use TensorFlow's attention implementation. The TensorFlow attention version is simpler and faster, but doesn't follow the settings in paper, whereas our implementation does.
- **multi_sparse_combiner** (*{'normal', 'mean', 'sum', 'sqrtn'}, default: 'sqrtn'*) – Options for combining *multi_sparse* features.
- **seed** (*int, default: 42*) – Random seed.
- **lower_upper_bound** (*tuple or None, default: None*) – Lower and upper score bound for *rating* task.
- **tf_sess_config** (*dict or None, default: None*) – Optional TensorFlow session config, see ConfigProto options.

References

Guorui Zhou et al. Deep Interest Network for Click-Through Rate Prediction.

fit(*train_data*, *verbose=1*, *shuffle=True*, *eval_data=None*, *metrics=None*, *k=10*, *eval_batch_size=8192*, *eval_user_num=None*)

Fit TF model on the training data.

Parameters

- **train_data** (*TransformedSet* object) – Data object used for training.
- **verbose** (*int, default: 1*) – Print verbosity. If *eval_data* is provided, setting it to higher than 1 will print evaluation metrics during training.
- **shuffle** (*bool, default: True*) – Whether to shuffle the training data.
- **eval_data** (*TransformedSet* object, default: None) – Data object used for evaluating.
- **metrics** (*list or None, default: None*) – List of metrics for evaluating.
- **k** (*int, default: 10*) – Parameter of metrics, e.g. recall at k, ndcg at k
- **eval_batch_size** (*int, default: 8192*) – Batch size for evaluating.

- **eval_user_num** (*int or None, default: None*) – Number of users for evaluating. Setting it to a positive number will sample users randomly from eval data.

Raises
RuntimeError – If *fit()* is called from a loaded model(*load()*).

classmethod load(*path*, *model_name*, *data_info*, *manual=True*)

Load saved TF model for inference.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.
- **data_info** (*DataInfo* object) – Object that contains some useful information.
- **manual** (*bool, default: True*) – Whether to load model variables using numpy. If you save the model using *manual*, you should also load the mode using *manual*.

Returns
model – Loaded TF model.

Return type
type(cls)

See also:

save

predict(*user*, *item*, *feats=None*, *cold_start='average'*, *inner_id=False*)

Make prediction(s) on given user(s) and item(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids.
- **item** (*int or str or array_like*) – Item id or batch of item ids.
- **feats** (*dict or pandas.Series or None, default: None*) – Extra features used in prediction.
- **cold_start** (*{'popular', 'average'}, default: 'average'*) – Cold start strategy.
  - 'popular' will sample from popular items.
  - 'average' will use the average of all the user/item embeddings as the representation of the cold-start user/item.
- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.

Returns
prediction – Predicted scores for each user-item pair.

Return type
float or numpy.ndarray

rebuild_model(*path*, *model_name*, *full_assign=True*)

Assign the saved model variables to the newly initialized model.

This method is used before retraining the new model, in order to avoid training from scratch every time we get some new data.

Parameters

- **path** (*str*) – File folder path for the saved model variables.
- **model_name** (*str*) – Name of the saved model file.
- **full_assign** (*bool, default: True*) – Whether to also restore the variables of Adam optimizer.

recommend_user(*user*, *n_rec*, *user_feats=None*, *item_data=None*, *cold_start='average'*, *inner_id=False*, *filter_consumed=True*, *random_rec=False*)

Recommend a list of items for given user(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids to recommend.
- **n_rec** (*int*) – Number of recommendations to return.
- **user_feats** (*dict or pandas.Series or None, default: None*) – Extra user features for recommendation.
- **item_data** (*pandas.DataFrame or None, default: None*) – Extra item features for recommendation.
- **cold_start** (*{'popular', 'average'}, default: 'average'*) – Cold start strategy.
  - 'popular' will sample from popular items.
  - 'average' will use the average of all the user/item embeddings as the representation of the cold-start user/item.
- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.
- **filter_consumed** (*bool, default: True*) – Whether to filter out items that a user has previously consumed.
- **random_rec** (*bool, default: False*) – Whether to choose items for recommendation based on their prediction scores.

Returns

recommendation – Recommendation result with user ids as keys and array_like recommended items as values.

Return type

dict of {Union[int, str, array_like] : numpy.ndarray}

save(*path*, *model_name*, *manual=True*, *inference_only=False*)

Save TF model for inference or retraining.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.
- **manual** (*bool, default: True*) – Whether to save model variables using numpy.
- **inference_only** (*bool, default: False*) – Whether to save model variables only for inference.

See also:

load

1.14.16 Item2Vec

class libreco.algorithms.**Item2Vec**(*task*, *data_info=None*, *embed_size=16*, *norm_embed=False*, *window_size=None*, *n_epochs=5*, *n_threads=0*, *seed=42*, *lower_upper_bound=None*)

Bases: *GensimBase*

Item2Vec algorithm.

> **Warning:** Item2Vec can only use in `ranking` task.

Parameters

- **task** (*{'ranking'}*) – Recommendation task. See *Task*.
- **data_info** (*DataInfo* object) – Object that contains useful information for training and inference.
- **embed_size** (*int, default: 16*) – Vector size of embeddings.
- **norm_embed** (*bool, default: False*) – Whether to normalize output embeddings.
- **window_size** (*int, default: 5*) – Maximum item distance within a sequence during training.
- **n_epochs** (*int, default: 10*) – Number of epochs for training.
- **n_threads** (*int, default: 0*) – Number of threads to use, *0* will use all cores.
- **seed** (*int, default: 42*) – Random seed.
- **lower_upper_bound** (*tuple or None, default: None*) – Lower and upper score bound for *rating* task.

References

Oren Barkan and Noam Koenigstein. Item2Vec: Neural Item Embedding for Collaborative Filtering.

fit(*train_data*, *verbose=1*, *shuffle=True*, *eval_data=None*, *metrics=None*, *k=10*, *eval_batch_size=8192*, *eval_user_num=None*)

Fit embed model on the training data.

Parameters

- **train_data** (*TransformedSet* object) – Data object used for training.
- **verbose** (*int, default: 1*) – Print verbosity. If *eval_data* is provided, setting it to higher than 1 will print evaluation metrics during training.
- **shuffle** (*bool, default: True*) – Whether to shuffle the training data.
- **eval_data** (*TransformedSet* object, default: None) – Data object used for evaluating.
- **metrics** (*list or None, default: None*) – List of metrics for evaluating.
- **k** (*int, default: 10*) – Parameter of metrics, e.g. recall at k, ndcg at k
- **eval_batch_size** (*int, default: 8192*) – Batch size for evaluating.

- **eval_user_num** (*int or None, default: None*) – Number of users for evaluating. Setting it to a positive number will sample users randomly from eval data.

Raises

RuntimeError – If *fit()* is called from a loaded model(*load()*).

get_item_embedding(*item=None*)

Get item embedding(s) from the model.

Parameters

item (*int or str or None*) – Query item id. If it is None, all item embeddings will be returned.

Returns

item_embedding – Returned item embeddings.

Return type

numpy.ndarray

Raises

- **ValueError** – If the item does not appear in the training data.
- **AssertionError** – If the model has not been trained.

get_user_embedding(*user=None*)

Get user embedding(s) from the model.

Parameters

user (*int or str or None*) – Query user id. If it is None, all user embeddings will be returned.

Returns

user_embedding – Returned user embeddings.

Return type

numpy.ndarray

Raises

- **ValueError** – If the user does not appear in the training data.
- **AssertionError** – If the model has not been trained.

init_knn(*approximate*, *sim_type*, *M=100*, *ef_construction=200*, *ef_search=200*)

Initialize k-nearest-search model.

Parameters

- **approximate** (*bool*) – Whether to use approximate nearest neighbor search. If it is True, nmslib must be installed. The *HNSW* method in *nmslib* is used.
- **sim_type** (*{'cosine', 'inner-product'}*) – Similarity space type.
- **M** (*int, default: 100*) – Parameter in *HNSW*, refer to nmslib doc.
- **ef_construction** (*int, default: 200*) – Parameter in *HNSW*, refer to nmslib doc.
- **ef_search** (*int, default: 200*) – Parameter in *HNSW*, refer to nmslib doc.

Raises

- **ValueError** – If sim_type is not one of ('cosine', 'inner-product').
- **ModuleNotFoundError** – If *approximate=True* and *nmslib* is not installed.

classmethod load(*path*, *model_name*, *data_info*, ***kwargs*)

Load saved embed model for inference.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.
- **data_info** (*DataInfo* object) – Object that contains some useful information.

Returns
model – Loaded embed model.

Return type
type(cls)

See also:

save

predict(*user*, *item*, *cold_start='average'*, *inner_id=False*)

Make prediction(s) on given user(s) and item(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids.
- **item** (*int or str or array_like*) – Item id or batch of item ids.
- **cold_start** (*{'popular', 'average'}, default: 'average'*) – Cold start strategy.
  - 'popular' will sample from popular items.
  - 'average' will use the average of all the user/item embeddings as the representation of the cold-start user/item.
- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.

Returns
prediction – Predicted scores for each user-item pair.

Return type
float or numpy.ndarray

rebuild_model(*path*, *model_name*)

Assign the saved model variables to the newly initialized model.

This method is used before retraining the new model, in order to avoid training from scratch every time we get some new data.

Parameters

- **path** (*str*) – File folder path for the saved model variables.
- **model_name** (*str*) – Name of the saved model file.

recommend_user(*user*, *n_rec*, *cold_start='average'*, *inner_id=False*, *filter_consumed=True*, *random_rec=False*)

Recommend a list of items for given user(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids to recommend.

- **n_rec** (*int*) – Number of recommendations to return.
- **cold_start** (*{'popular', 'average'}, default: 'average'*) – Cold start strategy.
  - 'popular' will sample from popular items.
  - 'average' will use the average of all the user/item embeddings as the representation of the cold-start user/item.
- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.
- **filter_consumed** (*bool, default: True*) – Whether to filter out items that a user has previously consumed.
- **random_rec** (*bool, default: False*) – Whether to choose items for recommendation based on their prediction scores.

Returns
recommendation – Recommendation result with user ids as keys and array_like recommended items as values.

Return type
dict of {Union[int, str, array_like] : numpy.ndarray}

save(*path*, *model_name*, *inference_only=False*, ***_*)

Save embed model for inference or retraining.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.
- **inference_only** (*bool, default: False*) – Whether to save model only for inference. If it is True, only embeddings will be saved. Otherwise, model variables will be saved.

See also:

load

search_knn_items(*item*, *k*)

Search most similar k items.

Parameters

- **item** (*int or str*) – Query item id.
- **k** (*int*) – Number of similar items.

Returns
similar items – A list of k similar items.

Return type
list

search_knn_users(*user*, *k*)

Search most similar k users.

Parameters

- **user** (*int or str*) – Query user id.
- **k** (*int*) – Number of similar users.

Returns
similar users – A list of k similar users.

Return type
list

1.14.17 RNN4Rec

class libreco.algorithms.**RNN4Rec**(*task*, *data_info=None*, *loss_type='cross_entropy'*, *rnn_type='gru'*, *embed_size=16*, *n_epochs=20*, *lr=0.001*, *lr_decay=False*, *epsilon=1e-05*, *reg=None*, *batch_size=256*, *num_neg=1*, *dropout_rate=None*, *hidden_units=16*, *use_layer_norm=False*, *recent_num=10*, *random_num=None*, *seed=42*, *lower_upper_bound=None*, *tf_sess_config=None*)

Bases: *EmbedBase*

RNN4Rec algorithm.

Note: The original paper used GRU, but in this implementation we can also use LSTM.

Parameters

- **task** (*{'rating', 'ranking'}*) – Recommendation task. See *Task*.
- **data_info** (*DataInfo* object) – Object that contains useful information for training and inference.
- **loss_type** (*{'cross_entropy', 'focal', 'bpr'}, default: 'cross_entropy'*) – Loss for model training.
- **rnn_type** (*{'lstm', 'gru'}, default: 'gru'*) – RNN for modeling.
- **embed_size** (*int, default: 16*) – Vector size of embeddings.
- **n_epochs** (*int, default: 10*) – Number of epochs for training.
- **lr** (*float, default 0.001*) – Learning rate for training.
- **lr_decay** (*bool, default: False*) – Whether to use learning rate decay.
- **epsilon** (*float, default: 1e-5*) – A small constant added to the denominator to improve numerical stability in Adam optimizer. According to the official comment, default value of *1e-8* for *epsilon* is generally not good, so here we choose *1e-5*. Users can try tuning this hyperparameter if the training is unstable.
- **reg** (*float or None, default: None*) – Regularization parameter, must be non-negative or None.
- **batch_size** (*int, default: 256*) – Batch size for training.
- **num_neg** (*int, default: 1*) – Number of negative samples for each positive sample, only used in *ranking* task.
- **dropout_rate** (*float or None, default: None*) – Probability of an element to be zeroed. If it is None, dropout is not used.
- **hidden_units** (*int, list of int or tuple of (int,), default: 16*) – Number of layers and corresponding layer size in RNN.

  Changed in version 1.0.0: Accept type of `int`, `list` or `tuple`, instead of `str`.

- **use_layer_norm** (*bool, default: False*) – Whether to use layer normalization.
- **recent_num** (*int or None, default: 10*) – Number of recent items to use in user behavior sequence.
- **random_num** (*int or None, default: None*) – Number of random sampled items to use in user behavior sequence. If *recent_num* is not None, *random_num* is not considered.
- **seed** (*int, default: 42*) – Random seed.
- **lower_upper_bound** (*tuple or None, default: None*) – Lower and upper score bound for *rating* task.
- **tf_sess_config** (*dict or None, default: None*) – Optional TensorFlow session config, see ConfigProto options.

References

Balazs Hidasi et al. Session-based Recommendations with Recurrent Neural Networks.

fit(*train_data*, *verbose=1*, *shuffle=True*, *eval_data=None*, *metrics=None*, *k=10*, *eval_batch_size=8192*, *eval_user_num=None*)

Fit embed model on the training data.

Parameters

- **train_data** (*TransformedSet* object) – Data object used for training.
- **verbose** (*int, default: 1*) – Print verbosity. If *eval_data* is provided, setting it to higher than 1 will print evaluation metrics during training.
- **shuffle** (*bool, default: True*) – Whether to shuffle the training data.
- **eval_data** (*TransformedSet* object, default: None) – Data object used for evaluating.
- **metrics** (*list or None, default: None*) – List of metrics for evaluating.
- **k** (*int, default: 10*) – Parameter of metrics, e.g. recall at k, ndcg at k
- **eval_batch_size** (*int, default: 8192*) – Batch size for evaluating.
- **eval_user_num** (*int or None, default: None*) – Number of users for evaluating. Setting it to a positive number will sample users randomly from eval data.

Raises

RuntimeError – If *fit()* is called from a loaded model(*load()*).

get_item_embedding(*item=None*)

Get item embedding(s) from the model.

Parameters

item (*int or str or None*) – Query item id. If it is None, all item embeddings will be returned.

Returns

item_embedding – Returned item embeddings.

Return type

numpy.ndarray

Raises

- **ValueError** – If the item does not appear in the training data.

- **AssertionError** – If the model has not been trained.

get_user_embedding(*user=None*)

Get user embedding(s) from the model.

Parameters

user (*int or str or None*) – Query user id. If it is None, all user embeddings will be returned.

Returns

user_embedding – Returned user embeddings.

Return type

numpy.ndarray

Raises

- **ValueError** – If the user does not appear in the training data.
- **AssertionError** – If the model has not been trained.

init_knn(*approximate*, *sim_type*, *M=100*, *ef_construction=200*, *ef_search=200*)

Initialize k-nearest-search model.

Parameters

- **approximate** (*bool*) – Whether to use approximate nearest neighbor search. If it is True, nmslib must be installed. The *HNSW* method in *nmslib* is used.
- **sim_type** (*{'cosine', 'inner-product'}*) – Similarity space type.
- **M** (*int, default: 100*) – Parameter in *HNSW*, refer to nmslib doc.
- **ef_construction** (*int, default: 200*) – Parameter in *HNSW*, refer to nmslib doc.
- **ef_search** (*int, default: 200*) – Parameter in *HNSW*, refer to nmslib doc.

Raises

- **ValueError** – If sim_type is not one of ('cosine', 'inner-product').
- **ModuleNotFoundError** – If *approximate=True* and *nmslib* is not installed.

classmethod load(*path*, *model_name*, *data_info*, ***kwargs*)

Load saved embed model for inference.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.
- **data_info** (*DataInfo* object) – Object that contains some useful information.

Returns

model – Loaded embed model.

Return type

type(cls)

See also:

save

predict(*user*, *item*, *cold_start='average'*, *inner_id=False*)

Make prediction(s) on given user(s) and item(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids.
- **item** (*int or str or array_like*) – Item id or batch of item ids.
- **cold_start** (*{'popular', 'average'}, default: 'average'*) – Cold start strategy.
  - 'popular' will sample from popular items.
  - 'average' will use the average of all the user/item embeddings as the representation of the cold-start user/item.
- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.

Returns

prediction – Predicted scores for each user-item pair.

Return type

float or numpy.ndarray

rebuild_model(*path*, *model_name*, *full_assign=True*)

Assign the saved model variables to the newly initialized model.

This method is used before retraining the new model, in order to avoid training from scratch every time we get some new data.

Parameters

- **path** (*str*) – File folder path for the saved model variables.
- **model_name** (*str*) – Name of the saved model file.
- **full_assign** (*bool, default: True*) – Whether to also restore the variables of Adam optimizer.

recommend_user(*user*, *n_rec*, *cold_start='average'*, *inner_id=False*, *filter_consumed=True*, *random_rec=False*)

Recommend a list of items for given user(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids to recommend.
- **n_rec** (*int*) – Number of recommendations to return.
- **cold_start** (*{'popular', 'average'}, default: 'average'*) – Cold start strategy.
  - 'popular' will sample from popular items.
  - 'average' will use the average of all the user/item embeddings as the representation of the cold-start user/item.
- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.
- **filter_consumed** (*bool, default: True*) – Whether to filter out items that a user has previously consumed.
- **random_rec** (*bool, default: False*) – Whether to choose items for recommendation based on their prediction scores.

Returns
recommendation – Recommendation result with user ids as keys and array_like recommended items as values.

Return type
dict of {Union[int, str, array_like] : numpy.ndarray}

save(*path*, *model_name*, *inference_only=False*, ***kwargs*)

Save embed model for inference or retraining.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.
- **inference_only** (*bool, default: False*) – Whether to save model only for inference. If it is True, only embeddings will be saved. Otherwise, model variables will be saved.

See also:

load

search_knn_items(*item*, *k*)

Search most similar k items.

Parameters

- **item** (*int or str*) – Query item id.
- **k** (*int*) – Number of similar items.

Returns
similar items – A list of k similar items.

Return type
list

search_knn_users(*user*, *k*)

Search most similar k users.

Parameters

- **user** (*int or str*) – Query user id.
- **k** (*int*) – Number of similar users.

Returns
similar users – A list of k similar users.

Return type
list

1.14.18 Caser

class libreco.algorithms.**Caser**(*task*, *data_info=None*, *loss_type='cross_entropy'*, *embed_size=16*, *n_epochs=20*, *lr=0.001*, *lr_decay=False*, *epsilon=1e-05*, *reg=None*, *batch_size=256*, *num_neg=1*, *use_bn=False*, *dropout_rate=None*, *nh_filters=2*, *nv_filters=4*, *recent_num=10*, *random_num=None*, *seed=42*, *lower_upper_bound=None*, *tf_sess_config=None*)

Bases: *EmbedBase*

Caser algorithm.

Parameters

- **task** (*{'rating', 'ranking'}*) – Recommendation task. See *Task*.
- **data_info** (*DataInfo* object) – Object that contains useful information for training and inference.
- **loss_type** (*{'cross_entropy', 'focal'}, default: 'cross_entropy'*) – Loss for model training.
- **embed_size** (*int, default: 16*) – Vector size of embeddings.
- **n_epochs** (*int, default: 10*) – Number of epochs for training.
- **lr** (*float, default 0.001*) – Learning rate for training.
- **lr_decay** (*bool, default: False*) – Whether to use learning rate decay.
- **epsilon** (*float, default: 1e-5*) – A small constant added to the denominator to improve numerical stability in Adam optimizer. According to the official comment, default value of *1e-8* for *epsilon* is generally not good, so here we choose *1e-5*. Users can try tuning this hyperparameter if the training is unstable.
- **reg** (*float or None, default: None*) – Regularization parameter, must be non-negative or None.
- **batch_size** (*int, default: 256*) – Batch size for training.
- **num_neg** (*int, default: 1*) – Number of negative samples for each positive sample, only used in *ranking* task.
- **use_bn** (*bool, default: True*) – Whether to use batch normalization.
- **dropout_rate** (*float or None, default: None*) – Probability of an element to be zeroed. If it is None, dropout is not used.
- **nh_filters** (*int, default: 2*) – Number of output filters in the horizontal CNN layer.
- **nv_filters** (*int, default: 4*) – Number of output filters in the vertical CNN layer.
- **recent_num** (*int or None, default: 10*) – Number of recent items to use in user behavior sequence.
- **random_num** (*int or None, default: None*) – Number of random sampled items to use in user behavior sequence. If *recent_num* is not None, *random_num* is not considered.
- **seed** (*int, default: 42*) – Random seed.
- **lower_upper_bound** (*tuple or None, default: None*) – Lower and upper score bound for *rating* task.
- **tf_sess_config** (*dict or None, default: None*) – Optional TensorFlow session config, see ConfigProto options.

References

Jiaxi Tang & Ke Wang. Personalized Top-N Sequential Recommendation via Convolutional Sequence Embedding.

fit(*train_data*, *verbose=1*, *shuffle=True*, *eval_data=None*, *metrics=None*, *k=10*, *eval_batch_size=8192*, *eval_user_num=None*)

Fit embed model on the training data.

Parameters

- **train_data** (*TransformedSet* object) – Data object used for training.
- **verbose** (*int, default: 1*) – Print verbosity. If *eval_data* is provided, setting it to higher than 1 will print evaluation metrics during training.
- **shuffle** (*bool, default: True*) – Whether to shuffle the training data.
- **eval_data** (*TransformedSet* object, default: None) – Data object used for evaluating.
- **metrics** (*list or None, default: None*) – List of metrics for evaluating.
- **k** (*int, default: 10*) – Parameter of metrics, e.g. recall at k, ndcg at k
- **eval_batch_size** (*int, default: 8192*) – Batch size for evaluating.
- **eval_user_num** (*int or None, default: None*) – Number of users for evaluating. Setting it to a positive number will sample users randomly from eval data.

Raises

RuntimeError – If *fit()* is called from a loaded model(*load()*).

get_item_embedding(*item=None*)

Get item embedding(s) from the model.

Parameters

item (*int or str or None*) – Query item id. If it is None, all item embeddings will be returned.

Returns

item_embedding – Returned item embeddings.

Return type

numpy.ndarray

Raises

- **ValueError** – If the item does not appear in the training data.
- **AssertionError** – If the model has not been trained.

get_user_embedding(*user=None*)

Get user embedding(s) from the model.

Parameters

user (*int or str or None*) – Query user id. If it is None, all user embeddings will be returned.

Returns

user_embedding – Returned user embeddings.

Return type

numpy.ndarray

Raises

- **ValueError** – If the user does not appear in the training data.
- **AssertionError** – If the model has not been trained.

init_knn(*approximate*, *sim_type*, *M=100*, *ef_construction=200*, *ef_search=200*)

Initialize k-nearest-search model.

Parameters

- **approximate** (*bool*) – Whether to use approximate nearest neighbor search. If it is True, nmslib must be installed. The *HNSW* method in *nmslib* is used.
- **sim_type** (*{'cosine', 'inner-product'}*) – Similarity space type.
- **M** (*int, default: 100*) – Parameter in *HNSW*, refer to nmslib doc.
- **ef_construction** (*int, default: 200*) – Parameter in *HNSW*, refer to nmslib doc.
- **ef_search** (*int, default: 200*) – Parameter in *HNSW*, refer to nmslib doc.

Raises

- **ValueError** – If sim_type is not one of ('cosine', 'inner-product').
- **ModuleNotFoundError** – If *approximate=True* and *nmslib* is not installed.

classmethod load(*path*, *model_name*, *data_info*, ***kwargs*)

Load saved embed model for inference.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.
- **data_info** (*DataInfo* object) – Object that contains some useful information.

Returns

model – Loaded embed model.

Return type

type(cls)

See also:

save

predict(*user*, *item*, *cold_start='average'*, *inner_id=False*)

Make prediction(s) on given user(s) and item(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids.
- **item** (*int or str or array_like*) – Item id or batch of item ids.
- **cold_start** (*{'popular', 'average'}, default: 'average'*) – Cold start strategy.
  - 'popular' will sample from popular items.
  - 'average' will use the average of all the user/item embeddings as the representation of the cold-start user/item.
- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.

Returns
prediction – Predicted scores for each user-item pair.

Return type
float or numpy.ndarray

rebuild_model(*path*, *model_name*, *full_assign=True*)

Assign the saved model variables to the newly initialized model.

This method is used before retraining the new model, in order to avoid training from scratch every time we get some new data.

Parameters

- **path** (*str*) – File folder path for the saved model variables.
- **model_name** (*str*) – Name of the saved model file.
- **full_assign** (*bool, default: True*) – Whether to also restore the variables of Adam optimizer.

recommend_user(*user*, *n_rec*, *cold_start='average'*, *inner_id=False*, *filter_consumed=True*, *random_rec=False*)

Recommend a list of items for given user(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids to recommend.
- **n_rec** (*int*) – Number of recommendations to return.
- **cold_start** (*{'popular', 'average'}, default: 'average'*) – Cold start strategy.
  - 'popular' will sample from popular items.
  - 'average' will use the average of all the user/item embeddings as the representation of the cold-start user/item.
- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.
- **filter_consumed** (*bool, default: True*) – Whether to filter out items that a user has previously consumed.
- **random_rec** (*bool, default: False*) – Whether to choose items for recommendation based on their prediction scores.

Returns
recommendation – Recommendation result with user ids as keys and array_like recommended items as values.

Return type
dict of {Union[int, str, array_like] : numpy.ndarray}

save(*path*, *model_name*, *inference_only=False*, ***kwargs*)

Save embed model for inference or retraining.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.

- **inference_only** (*bool, default: False*) – Whether to save model only for inference. If it is True, only embeddings will be saved. Otherwise, model variables will be saved.

See also:

load

search_knn_items(*item*, *k*)

Search most similar k items.

Parameters

- **item** (*int or str*) – Query item id.
- **k** (*int*) – Number of similar items.

Returns
similar items – A list of k similar items.

Return type
list

search_knn_users(*user*, *k*)

Search most similar k users.

Parameters

- **user** (*int or str*) – Query user id.
- **k** (*int*) – Number of similar users.

Returns
similar users – A list of k similar users.

Return type
list

1.14.19 WaveNet

class libreco.algorithms.**WaveNet**(*task*, *data_info=None*, *loss_type='cross_entropy'*, *embed_size=16*, *n_epochs=20*, *lr=0.001*, *lr_decay=False*, *epsilon=1e-05*, *reg=None*, *batch_size=256*, *num_neg=1*, *dropout_rate=None*, *use_bn=False*, *n_filters=16*, *n_blocks=1*, *n_layers_per_block=4*, *recent_num=10*, *random_num=None*, *seed=42*, *lower_upper_bound=None*, *tf_sess_config=None*)

Bases: *EmbedBase*

WaveNet algorithm.

Parameters

- **task** (*{'rating', 'ranking'}*) – Recommendation task. See *Task*.
- **data_info** (*DataInfo* object) – Object that contains useful information for training and inference.
- **loss_type** (*{'cross_entropy', 'focal'}, default: 'cross_entropy'*) – Loss for model training.
- **embed_size** (*int, default: 16*) – Vector size of embeddings.
- **n_epochs** (*int, default: 10*) – Number of epochs for training.

- **lr** (*float, default 0.001*) – Learning rate for training.
- **lr_decay** (*bool, default: False*) – Whether to use learning rate decay.
- **epsilon** (*float, default: 1e-5*) – A small constant added to the denominator to improve numerical stability in Adam optimizer. According to the official comment, default value of *1e-8* for *epsilon* is generally not good, so here we choose *1e-5*. Users can try tuning this hyperparameter if the training is unstable.
- **reg** (*float or None, default: None*) – Regularization parameter, must be non-negative or None.
- **batch_size** (*int, default: 256*) – Batch size for training.
- **num_neg** (*int, default: 1*) – Number of negative samples for each positive sample, only used in *ranking* task.
- **use_bn** (*bool, default: True*) – Whether to use batch normalization.
- **dropout_rate** (*float or None, default: None*) – Probability of an element to be zeroed. If it is None, dropout is not used.
- **n_filters** (*int, default: 16*) – Number of output filters in each CNN layer.
- **n_blocks** (*int, default: 1*) – Number of CNN blocks.
- **n_layers_per_block** (*int, default: 4*) – Number of CNN layers in each block.
- **recent_num** (*int or None, default: 10*) – Number of recent items to use in user behavior sequence.
- **random_num** (*int or None, default: None*) – Number of random sampled items to use in user behavior sequence. If *recent_num* is not None, *random_num* is not considered.
- **seed** (*int, default: 42*) – Random seed.
- **lower_upper_bound** (*tuple or None, default: None*) – Lower and upper score bound for *rating* task.
- **tf_sess_config** (*dict or None, default: None*) – Optional TensorFlow session config, see ConfigProto options.

References

Aaron van den Oord et al. WaveNet: A Generative Model for Raw Audio.

fit(*train_data*, *verbose=1*, *shuffle=True*, *eval_data=None*, *metrics=None*, *k=10*, *eval_batch_size=8192*, *eval_user_num=None*)

Fit embed model on the training data.

Parameters

- **train_data** (*TransformedSet* object) – Data object used for training.
- **verbose** (*int, default: 1*) – Print verbosity. If *eval_data* is provided, setting it to higher than 1 will print evaluation metrics during training.
- **shuffle** (*bool, default: True*) – Whether to shuffle the training data.
- **eval_data** (*TransformedSet* object, default: None) – Data object used for evaluating.
- **metrics** (*list or None, default: None*) – List of metrics for evaluating.
- **k** (*int, default: 10*) – Parameter of metrics, e.g. recall at k, ndcg at k

- **eval_batch_size** (*int, default: 8192*) – Batch size for evaluating.
- **eval_user_num** (*int or None, default: None*) – Number of users for evaluating. Setting it to a positive number will sample users randomly from eval data.

Raises
RuntimeError – If *fit()* is called from a loaded model(*load()*).

get_item_embedding(*item=None*)

Get item embedding(s) from the model.

Parameters
item (*int or str or None*) – Query item id. If it is None, all item embeddings will be returned.

Returns
item_embedding – Returned item embeddings.

Return type
numpy.ndarray

Raises
- ValueError – If the item does not appear in the training data.
- AssertionError – If the model has not been trained.

get_user_embedding(*user=None*)

Get user embedding(s) from the model.

Parameters
user (*int or str or None*) – Query user id. If it is None, all user embeddings will be returned.

Returns
user_embedding – Returned user embeddings.

Return type
numpy.ndarray

Raises
- ValueError – If the user does not appear in the training data.
- AssertionError – If the model has not been trained.

init_knn(*approximate*, *sim_type*, *M=100*, *ef_construction=200*, *ef_search=200*)

Initialize k-nearest-search model.

Parameters
- **approximate** (*bool*) – Whether to use approximate nearest neighbor search. If it is True, nmslib must be installed. The *HNSW* method in *nmslib* is used.
- **sim_type** (*{'cosine', 'inner-product'}*) – Similarity space type.
- **M** (*int, default: 100*) – Parameter in *HNSW*, refer to nmslib doc.
- **ef_construction** (*int, default: 200*) – Parameter in *HNSW*, refer to nmslib doc.
- **ef_search** (*int, default: 200*) – Parameter in *HNSW*, refer to nmslib doc.

Raises
- ValueError – If sim_type is not one of ('cosine', 'inner-product').

- **ModuleNotFoundError** – If *approximate=True* and *nmslib* is not installed.

classmethod load(*path*, *model_name*, *data_info*, ***kwargs*)

Load saved embed model for inference.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.
- **data_info** (*DataInfo* object) – Object that contains some useful information.

Returns
model – Loaded embed model.

Return type
type(cls)

See also:

save

predict(*user*, *item*, *cold_start='average'*, *inner_id=False*)

Make prediction(s) on given user(s) and item(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids.
- **item** (*int or str or array_like*) – Item id or batch of item ids.
- **cold_start** (*{'popular', 'average'}, default: 'average'*) – Cold start strategy.
  - 'popular' will sample from popular items.
  - 'average' will use the average of all the user/item embeddings as the representation of the cold-start user/item.
- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.

Returns
prediction – Predicted scores for each user-item pair.

Return type
float or numpy.ndarray

rebuild_model(*path*, *model_name*, *full_assign=True*)

Assign the saved model variables to the newly initialized model.

This method is used before retraining the new model, in order to avoid training from scratch every time we get some new data.

Parameters

- **path** (*str*) – File folder path for the saved model variables.
- **model_name** (*str*) – Name of the saved model file.
- **full_assign** (*bool, default: True*) – Whether to also restore the variables of Adam optimizer.

recommend_user(*user*, *n_rec*, *cold_start='average'*, *inner_id=False*, *filter_consumed=True*, *random_rec=False*)

Recommend a list of items for given user(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids to recommend.
- **n_rec** (*int*) – Number of recommendations to return.
- **cold_start** (*{'popular', 'average'}, default: 'average'*) – Cold start strategy.
  - 'popular' will sample from popular items.
  - 'average' will use the average of all the user/item embeddings as the representation of the cold-start user/item.
- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.
- **filter_consumed** (*bool, default: True*) – Whether to filter out items that a user has previously consumed.
- **random_rec** (*bool, default: False*) – Whether to choose items for recommendation based on their prediction scores.

Returns

recommendation – Recommendation result with user ids as keys and array_like recommended items as values.

Return type

dict of {Union[int, str, array_like] : numpy.ndarray}

save(*path*, *model_name*, *inference_only=False*, ***kwargs*)

Save embed model for inference or retraining.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.
- **inference_only** (*bool, default: False*) – Whether to save model only for inference. If it is True, only embeddings will be saved. Otherwise, model variables will be saved.

See also:

load

search_knn_items(*item*, *k*)

Search most similar k items.

Parameters

- **item** (*int or str*) – Query item id.
- **k** (*int*) – Number of similar items.

Returns

similar items – A list of k similar items.

Return type

list

search_knn_users(*user*, *k*)

Search most similar k users.

Parameters

- **user** (*int or str*) – Query user id.
- **k** (*int*) – Number of similar users.

Returns

similar users – A list of k similar users.

Return type

list

1.14.20 DeepWalk

class libreco.algorithms.**DeepWalk**(*task*, *data_info*, *embed_size=16*, *norm_embed=False*, *n_walks=10*, *walk_length=10*, *window_size=5*, *n_epochs=5*, *n_threads=0*, *seed=42*, *lower_upper_bound=None*)

Bases: *GensimBase*

DeepWalk algorithm.

> **Caution:** DeepWalk can only use in `ranking` task.

Parameters

- **task** (*{'ranking'}*) – Recommendation task. See *Task*.
- **data_info** (*DataInfo* object) – Object that contains useful information for training and inference.
- **embed_size** (*int, default: 16*) – Vector size of embeddings.
- **norm_embed** (*bool, default: False*) – Whether to normalize output embeddings.
- **n_walks** (*int, default: 10*) – Number of walks for every item.
- **walk_length** (*int, default: 10*) – Length of each walk.
- **window_size** (*int, default: 5*) – Maximum item distance within a sequence during training.
- **n_epochs** (*int, default: 10*) – Number of epochs for training.
- **n_threads** (*int, default: 0*) – Number of threads to use, *0* will use all cores.
- **seed** (*int, default: 42*) – Random seed.
- **lower_upper_bound** (*tuple or None, default: None*) – Lower and upper score bound for *rating* task.

References

Bryan Perozzi et al. DeepWalk: Online Learning of Social Representations.

fit(*train_data*, *verbose=1*, *shuffle=True*, *eval_data=None*, *metrics=None*, *k=10*, *eval_batch_size=8192*, *eval_user_num=None*)

Fit embed model on the training data.

Parameters

- **train_data** (*TransformedSet* object) – Data object used for training.
- **verbose** (*int, default: 1*) – Print verbosity. If *eval_data* is provided, setting it to higher than 1 will print evaluation metrics during training.
- **shuffle** (*bool, default: True*) – Whether to shuffle the training data.
- **eval_data** (*TransformedSet* object, default: None) – Data object used for evaluating.
- **metrics** (*list or None, default: None*) – List of metrics for evaluating.
- **k** (*int, default: 10*) – Parameter of metrics, e.g. recall at k, ndcg at k
- **eval_batch_size** (*int, default: 8192*) – Batch size for evaluating.
- **eval_user_num** (*int or None, default: None*) – Number of users for evaluating. Setting it to a positive number will sample users randomly from eval data.

Raises

RuntimeError – If *fit()* is called from a loaded model(*load()*).

get_item_embedding(*item=None*)

Get item embedding(s) from the model.

Parameters

item (*int or str or None*) – Query item id. If it is None, all item embeddings will be returned.

Returns

item_embedding – Returned item embeddings.

Return type

numpy.ndarray

Raises

- **ValueError** – If the item does not appear in the training data.
- **AssertionError** – If the model has not been trained.

get_user_embedding(*user=None*)

Get user embedding(s) from the model.

Parameters

user (*int or str or None*) – Query user id. If it is None, all user embeddings will be returned.

Returns

user_embedding – Returned user embeddings.

Return type

numpy.ndarray

Raises

- **ValueError** – If the user does not appear in the training data.
- **AssertionError** – If the model has not been trained.

init_knn(*approximate*, *sim_type*, *M=100*, *ef_construction=200*, *ef_search=200*)

Initialize k-nearest-search model.

Parameters

- **approximate** (*bool*) – Whether to use approximate nearest neighbor search. If it is True, nmslib must be installed. The *HNSW* method in *nmslib* is used.
- **sim_type** (*{'cosine', 'inner-product'}*) – Similarity space type.
- **M** (*int, default: 100*) – Parameter in *HNSW*, refer to nmslib doc.
- **ef_construction** (*int, default: 200*) – Parameter in *HNSW*, refer to nmslib doc.
- **ef_search** (*int, default: 200*) – Parameter in *HNSW*, refer to nmslib doc.

Raises

- **ValueError** – If sim_type is not one of ('cosine', 'inner-product').
- **ModuleNotFoundError** – If *approximate=True* and *nmslib* is not installed.

classmethod load(*path*, *model_name*, *data_info*, ***kwargs*)

Load saved embed model for inference.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.
- **data_info** (*DataInfo* object) – Object that contains some useful information.

Returns

model – Loaded embed model.

Return type

type(cls)

See also:

save

predict(*user*, *item*, *cold_start='average'*, *inner_id=False*)

Make prediction(s) on given user(s) and item(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids.
- **item** (*int or str or array_like*) – Item id or batch of item ids.
- **cold_start** (*{'popular', 'average'}, default: 'average'*) – Cold start strategy.
  - 'popular' will sample from popular items.
  - 'average' will use the average of all the user/item embeddings as the representation of the cold-start user/item.
- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.

Returns

prediction – Predicted scores for each user-item pair.

Return type

float or numpy.ndarray

rebuild_model(*path*, *model_name*)

Assign the saved model variables to the newly initialized model.

This method is used before retraining the new model, in order to avoid training from scratch every time we get some new data.

Parameters

- **path** (*str*) – File folder path for the saved model variables.
- **model_name** (*str*) – Name of the saved model file.

recommend_user(*user*, *n_rec*, *cold_start='average'*, *inner_id=False*, *filter_consumed=True*, *random_rec=False*)

Recommend a list of items for given user(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids to recommend.
- **n_rec** (*int*) – Number of recommendations to return.
- **cold_start** (*{'popular', 'average'}, default: 'average'*) – Cold start strategy.
  - 'popular' will sample from popular items.
  - 'average' will use the average of all the user/item embeddings as the representation of the cold-start user/item.
- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.
- **filter_consumed** (*bool, default: True*) – Whether to filter out items that a user has previously consumed.
- **random_rec** (*bool, default: False*) – Whether to choose items for recommendation based on their prediction scores.

Returns

recommendation – Recommendation result with user ids as keys and array_like recommended items as values.

Return type

dict of {Union[int, str, array_like] : numpy.ndarray}

save(*path*, *model_name*, *inference_only=False*, ***_)

Save embed model for inference or retraining.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.
- **inference_only** (*bool, default: False*) – Whether to save model only for inference. If it is True, only embeddings will be saved. Otherwise, model variables will be saved.

See also:

load

search_knn_items(*item*, *k*)

Search most similar k items.

Parameters

- **item** (*int or str*) – Query item id.
- **k** (*int*) – Number of similar items.

Returns
similar items – A list of k similar items.

Return type
list

search_knn_users(*user*, *k*)

Search most similar k users.

Parameters

- **user** (*int or str*) – Query user id.
- **k** (*int*) – Number of similar users.

Returns
similar users – A list of k similar users.

Return type
list

1.14.21 NGCF

class libreco.algorithms.**NGCF**(*task*, *data_info*, *loss_type='cross_entropy'*, *embed_size=16*, *n_epochs=20*, *lr=0.001*, *lr_decay=False*, *epsilon=1e-08*, *amsgrad=False*, *reg=None*, *batch_size=256*, *num_neg=1*, *node_dropout=0.0*, *message_dropout=0.0*, *hidden_units=(64, 64, 64)*, *margin=1.0*, *sampler='random'*, *seed=42*, *device='cuda'*, *lower_upper_bound=None*)

Bases: *EmbedBase*

Neural Graph Collaborative Filtering algorithm.

> **Warning:** NGCF can only be used in `ranking` task.

Parameters

- **task** (*{'ranking'}*) – Recommendation task. See *Task*.
- **data_info** (*DataInfo* object) – Object that contains useful information for training and inference.
- **loss_type** (*{'cross_entropy', 'focal', 'bpr', 'max_margin'}, default: 'cross_entropy'*) – Loss for model training.
- **embed_size** (*int, default: 16*) – Vector size of embeddings.
- **n_epochs** (*int, default: 10*) – Number of epochs for training.
- **lr** (*float, default 0.001*) – Learning rate for training.
- **lr_decay** (*bool, default: False*) – Whether to use learning rate decay.
- **epsilon** (*float, default: 1e-8*) – A small constant added to the denominator to improve numerical stability in Adam optimizer.

- **amsgrad** (*bool, default: False*) – Whether to use the AMSGrad variant from the paper On the Convergence of Adam and Beyond.
- **reg** (*float or None, default: None*) – Regularization parameter, must be non-negative or None.
- **batch_size** (*int, default: 256*) – Batch size for training.
- **num_neg** (*int, default: 1*) – Number of negative samples for each positive sample.
- **node_dropout** (*float, default: 0.0*) – Node dropout probability. 0.0 means node dropout is not used.
- **message_dropout** (*float, default: 0.0*) – Message dropout probability. 0.0 means message dropout is not used.
- **hidden_units** (*int, list of int or tuple of (int,), default: (64, 64, 64)*) – Number of layers and corresponding layer size in embedding propagation.
- **margin** (*float, default: 1.0*) – Margin used in *max_margin* loss.
- **sampler** (*{'random', 'unconsumed', 'popular'}, default: 'random'*) – Negative sampling strategy.
  - `'random'` means random sampling.
  - `'unconsumed'` samples items that the target user did not consume before.
  - `'popular'` has a higher probability to sample popular items as negative samples.
- **seed** (*int, default: 42*) – Random seed.
- **device** (*{'cpu', 'cuda'}, default: 'cuda'*) – Refer to torch.device.

  Changed in version 1.0.0: Accept str type `'cpu'` or `'cuda'`, instead of `torch.device(..)`.
- **lower_upper_bound** (*tuple or None, default: None*) – Lower and upper score bound for *rating* task.

References

Xiang Wang et al. Neural Graph Collaborative Filtering.

fit(*train_data*, *verbose=1*, *shuffle=True*, *eval_data=None*, *metrics=None*, *k=10*, *eval_batch_size=8192*, *eval_user_num=None*)

Fit embed model on the training data.

Parameters

- **train_data** (*TransformedSet* object) – Data object used for training.
- **verbose** (*int, default: 1*) – Print verbosity. If *eval_data* is provided, setting it to higher than 1 will print evaluation metrics during training.
- **shuffle** (*bool, default: True*) – Whether to shuffle the training data.
- **eval_data** (*TransformedSet* object, default: None) – Data object used for evaluating.
- **metrics** (*list or None, default: None*) – List of metrics for evaluating.
- **k** (*int, default: 10*) – Parameter of metrics, e.g. recall at k, ndcg at k
- **eval_batch_size** (*int, default: 8192*) – Batch size for evaluating.

- **eval_user_num** (*int or None, default: None*) – Number of users for evaluating. Setting it to a positive number will sample users randomly from eval data.

Raises

RuntimeError – If *fit()* is called from a loaded model(*load()*).

get_item_embedding(*item=None*)

Get item embedding(s) from the model.

Parameters

item (*int or str or None*) – Query item id. If it is None, all item embeddings will be returned.

Returns

item_embedding – Returned item embeddings.

Return type

numpy.ndarray

Raises

- **ValueError** – If the item does not appear in the training data.
- **AssertionError** – If the model has not been trained.

get_user_embedding(*user=None*)

Get user embedding(s) from the model.

Parameters

user (*int or str or None*) – Query user id. If it is None, all user embeddings will be returned.

Returns

user_embedding – Returned user embeddings.

Return type

numpy.ndarray

Raises

- **ValueError** – If the user does not appear in the training data.
- **AssertionError** – If the model has not been trained.

init_knn(*approximate*, *sim_type*, *M=100*, *ef_construction=200*, *ef_search=200*)

Initialize k-nearest-search model.

Parameters

- **approximate** (*bool*) – Whether to use approximate nearest neighbor search. If it is True, nmslib must be installed. The *HNSW* method in *nmslib* is used.
- **sim_type** (*{'cosine', 'inner-product'}*) – Similarity space type.
- **M** (*int, default: 100*) – Parameter in *HNSW*, refer to nmslib doc.
- **ef_construction** (*int, default: 200*) – Parameter in *HNSW*, refer to nmslib doc.
- **ef_search** (*int, default: 200*) – Parameter in *HNSW*, refer to nmslib doc.

Raises

- **ValueError** – If sim_type is not one of ('cosine', 'inner-product').
- **ModuleNotFoundError** – If *approximate=True* and *nmslib* is not installed.

classmethod load(*path*, *model_name*, *data_info*, ***kwargs*)

Load saved embed model for inference.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.
- **data_info** (*DataInfo* object) – Object that contains some useful information.

Returns
model – Loaded embed model.

Return type
type(cls)

See also:

save

predict(*user*, *item*, *cold_start='average'*, *inner_id=False*)

Make prediction(s) on given user(s) and item(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids.
- **item** (*int or str or array_like*) – Item id or batch of item ids.
- **cold_start** (*{'popular', 'average'}, default: 'average'*) – Cold start strategy.
  - 'popular' will sample from popular items.
  - 'average' will use the average of all the user/item embeddings as the representation of the cold-start user/item.
- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.

Returns
prediction – Predicted scores for each user-item pair.

Return type
float or numpy.ndarray

rebuild_model(*path*, *model_name*)

Assign the saved model variables to the newly initialized model.

This method is used before retraining the new model, in order to avoid training from scratch every time we get some new data.

Parameters

- **path** (*str*) – File folder path for the saved model variables.
- **model_name** (*str*) – Name of the saved model file.

recommend_user(*user*, *n_rec*, *cold_start='average'*, *inner_id=False*, *filter_consumed=True*, *random_rec=False*)

Recommend a list of items for given user(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids to recommend.

- **n_rec** (*int*) – Number of recommendations to return.
- **cold_start** (*{'popular', 'average'}, default: 'average'*) – Cold start strategy.
  - 'popular' will sample from popular items.
  - 'average' will use the average of all the user/item embeddings as the representation of the cold-start user/item.
- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.
- **filter_consumed** (*bool, default: True*) – Whether to filter out items that a user has previously consumed.
- **random_rec** (*bool, default: False*) – Whether to choose items for recommendation based on their prediction scores.

Returns
recommendation – Recommendation result with user ids as keys and array_like recommended items as values.

Return type
dict of {Union[int, str, array_like] : numpy.ndarray}

save(*path*, *model_name*, *inference_only=False*, ***kwargs*)

Save embed model for inference or retraining.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.
- **inference_only** (*bool, default: False*) – Whether to save model only for inference. If it is True, only embeddings will be saved. Otherwise, model variables will be saved.

See also:

load

search_knn_items(*item*, *k*)

Search most similar k items.

Parameters

- **item** (*int or str*) – Query item id.
- **k** (*int*) – Number of similar items.

Returns
similar items – A list of k similar items.

Return type
list

search_knn_users(*user*, *k*)

Search most similar k users.

Parameters

- **user** (*int or str*) – Query user id.
- **k** (*int*) – Number of similar users.

Returns
similar users – A list of k similar users.

Return type
list

1.14.22 LightGCN

class libreco.algorithms.**LightGCN**(*task*, *data_info*, *loss_type='bpr'*, *embed_size=16*, *n_epochs=20*, *lr=0.001*, *lr_decay=False*, *epsilon=1e-08*, *amsgrad=False*, *reg=None*, *batch_size=256*, *num_neg=1*, *dropout_rate=0.0*, *n_layers=3*, *margin=1.0*, *sampler='random'*, *seed=42*, *device='cuda'*, *lower_upper_bound=None*, *with_training=True*)

Bases: *EmbedBase*

LightGCN algorithm.

Caution: LightGCN can only be used in `ranking` task.

Parameters

- **task** (*{'ranking'}*) – Recommendation task. See *Task*.
- **data_info** (*DataInfo* object) – Object that contains useful information for training and inference.
- **loss_type** (*{'cross_entropy', 'focal', 'bpr', 'max_margin'}, default: 'bpr'*) – Loss for model training.
- **embed_size** (*int, default: 16*) – Vector size of embeddings.
- **n_epochs** (*int, default: 10*) – Number of epochs for training.
- **lr** (*float, default 0.001*) – Learning rate for training.
- **lr_decay** (*bool, default: False*) – Whether to use learning rate decay.
- **epsilon** (*float, default: 1e-8*) – A small constant added to the denominator to improve numerical stability in Adam optimizer.
- **amsgrad** (*bool, default: False*) – Whether to use the AMSGrad variant from the paper On the Convergence of Adam and Beyond.
- **reg** (*float or None, default: None*) – Regularization parameter, must be non-negative or None.
- **batch_size** (*int, default: 256*) – Batch size for training.
- **num_neg** (*int, default: 1*) – Number of negative samples for each positive sample.
- **dropout_rate** (*float, default: 0.0*) – Probability of a node being dropped. 0.0 means dropout is not used.
- **n_layers** (*int, default: 3*) – Number of GCN layer.
- **margin** (*float, default: 1.0*) – Margin used in *max_margin* loss.
- **sampler** (*{'random', 'unconsumed', 'popular'}, default: 'random'*) – Negative sampling strategy.
  - `'random'` means random sampling.

  - `'unconsumed'` samples items that the target user did not consume before.
  - `'popular'` has a higher probability to sample popular items as negative samples.
- **seed** (*int, default: 42*) – Random seed.
- **device** (*{'cpu', 'cuda'}, default: 'cuda'*) – Refer to torch.device.

  Changed in version 1.0.0: Accept str type `'cpu'` or `'cuda'`, instead of `torch.device(..)`.
- **lower_upper_bound** (*tuple or None, default: None*) – Lower and upper score bound for *rating* task.

References

Xiangnan He et al. LightGCN: Simplifying and Powering Graph Convolution Network for Recommendation.

fit(*train_data*, *verbose=1*, *shuffle=True*, *eval_data=None*, *metrics=None*, *k=10*, *eval_batch_size=8192*, *eval_user_num=None*)

Fit embed model on the training data.

Parameters

- **train_data** (*TransformedSet* object) – Data object used for training.
- **verbose** (*int, default: 1*) – Print verbosity. If *eval_data* is provided, setting it to higher than 1 will print evaluation metrics during training.
- **shuffle** (*bool, default: True*) – Whether to shuffle the training data.
- **eval_data** (*TransformedSet* object, default: None) – Data object used for evaluating.
- **metrics** (*list or None, default: None*) – List of metrics for evaluating.
- **k** (*int, default: 10*) – Parameter of metrics, e.g. recall at k, ndcg at k
- **eval_batch_size** (*int, default: 8192*) – Batch size for evaluating.
- **eval_user_num** (*int or None, default: None*) – Number of users for evaluating. Setting it to a positive number will sample users randomly from eval data.

Raises

RuntimeError – If *fit()* is called from a loaded model(*load()*).

get_item_embedding(*item=None*)

Get item embedding(s) from the model.

Parameters

item (*int or str or None*) – Query item id. If it is None, all item embeddings will be returned.

Returns

item_embedding – Returned item embeddings.

Return type

numpy.ndarray

Raises

- **ValueError** – If the item does not appear in the training data.
- **AssertionError** – If the model has not been trained.

get_user_embedding(*user=None*)

Get user embedding(s) from the model.

Parameters
user (*int or str or None*) – Query user id. If it is None, all user embeddings will be returned.

Returns
user_embedding – Returned user embeddings.

Return type
numpy.ndarray

Raises

- **ValueError** – If the user does not appear in the training data.
- **AssertionError** – If the model has not been trained.

init_knn(*approximate*, *sim_type*, *M=100*, *ef_construction=200*, *ef_search=200*)

Initialize k-nearest-search model.

Parameters

- **approximate** (*bool*) – Whether to use approximate nearest neighbor search. If it is True, nmslib must be installed. The *HNSW* method in *nmslib* is used.
- **sim_type** (*{'cosine', 'inner-product'}*) – Similarity space type.
- **M** (*int, default: 100*) – Parameter in *HNSW*, refer to nmslib doc.
- **ef_construction** (*int, default: 200*) – Parameter in *HNSW*, refer to nmslib doc.
- **ef_search** (*int, default: 200*) – Parameter in *HNSW*, refer to nmslib doc.

Raises

- **ValueError** – If sim_type is not one of ('cosine', 'inner-product').
- **ModuleNotFoundError** – If *approximate=True* and *nmslib* is not installed.

classmethod load(*path*, *model_name*, *data_info*, ***kwargs*)

Load saved embed model for inference.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.
- **data_info** (*DataInfo* object) – Object that contains some useful information.

Returns
model – Loaded embed model.

Return type
type(cls)

See also:

save

predict(*user*, *item*, *cold_start='average'*, *inner_id=False*)

Make prediction(s) on given user(s) and item(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids.
- **item** (*int or str or array_like*) – Item id or batch of item ids.
- **cold_start** (*{'popular', 'average'}, default: 'average'*) – Cold start strategy.
  - 'popular' will sample from popular items.
  - 'average' will use the average of all the user/item embeddings as the representation of the cold-start user/item.
- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.

Returns
prediction – Predicted scores for each user-item pair.

Return type
float or numpy.ndarray

rebuild_model(*path*, *model_name*)

Assign the saved model variables to the newly initialized model.

This method is used before retraining the new model, in order to avoid training from scratch every time we get some new data.

Parameters

- **path** (*str*) – File folder path for the saved model variables.
- **model_name** (*str*) – Name of the saved model file.

recommend_user(*user*, *n_rec*, *cold_start='average'*, *inner_id=False*, *filter_consumed=True*, *random_rec=False*)

Recommend a list of items for given user(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids to recommend.
- **n_rec** (*int*) – Number of recommendations to return.
- **cold_start** (*{'popular', 'average'}, default: 'average'*) – Cold start strategy.
  - 'popular' will sample from popular items.
  - 'average' will use the average of all the user/item embeddings as the representation of the cold-start user/item.
- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.
- **filter_consumed** (*bool, default: True*) – Whether to filter out items that a user has previously consumed.
- **random_rec** (*bool, default: False*) – Whether to choose items for recommendation based on their prediction scores.

Returns
recommendation – Recommendation result with user ids as keys and array_like recommended items as values.

Return type
dict of {Union[int, str, array_like] : numpy.ndarray}

save(*path*, *model_name*, *inference_only=False*, ***kwargs*)

Save embed model for inference or retraining.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.
- **inference_only** (*bool, default: False*) – Whether to save model only for inference. If it is True, only embeddings will be saved. Otherwise, model variables will be saved.

See also:

load

search_knn_items(*item*, *k*)

Search most similar k items.

Parameters

- **item** (*int or str*) – Query item id.
- **k** (*int*) – Number of similar items.

Returns

similar items – A list of k similar items.

Return type

list

search_knn_users(*user*, *k*)

Search most similar k users.

Parameters

- **user** (*int or str*) – Query user id.
- **k** (*int*) – Number of similar users.

Returns

similar users – A list of k similar users.

Return type

list

1.14.23 GraphSage

class libreco.algorithms.**GraphSage**(*task*, *data_info*, *loss_type='cross_entropy'*, *paradigm='i2i'*, *embed_size=16*, *n_epochs=20*, *lr=0.001*, *lr_decay=False*, *epsilon=1e-08*, *amsgrad=False*, *reg=None*, *batch_size=256*, *num_neg=1*, *dropout_rate=0.0*, *remove_edges=False*, *num_layers=2*, *num_neighbors=3*, *num_walks=10*, *sample_walk_len=5*, *margin=1.0*, *sampler='random'*, *start_node='random'*, *focus_start=False*, *seed=42*, *device='cuda'*, *lower_upper_bound=None*)

Bases: *EmbedBase*

GraphSage algorithm.

Note: This algorithm is implemented in PyTorch.

Caution: GraphSage can only be used in `ranking` task.

New in version 0.12.0.

Parameters

- **task** (*{'ranking'}*) – Recommendation task. See *Task*.
- **data_info** (*DataInfo* object) – Object that contains useful information for training and inference.
- **loss_type** (*{'cross_entropy', 'focal', 'bpr', 'max_margin'}, default: 'cross_entropy'*) – Loss for model training.
- **paradigm** (*{'u2i', 'i2i'}, default: 'i2i'*) – Choice for features in model.
  - `'u2i'` will combine user features and item features.
  - `'i2i'` will only use item features, this is the setting in the original paper.
- **embed_size** (*int, default: 16*) – Vector size of embeddings.
- **n_epochs** (*int, default: 10*) – Number of epochs for training.
- **lr** (*float, default 0.001*) – Learning rate for training.
- **lr_decay** (*bool, default: False*) – Whether to use learning rate decay.
- **epsilon** (*float, default: 1e-8*) – A small constant added to the denominator to improve numerical stability in Adam optimizer.
- **amsgrad** (*bool, default: False*) – Whether to use the AMSGrad variant from the paper On the Convergence of Adam and Beyond.
- **reg** (*float or None, default: None*) – Regularization parameter, must be non-negative or None.
- **batch_size** (*int, default: 256*) – Batch size for training.
- **num_neg** (*int, default: 1*) – Number of negative samples for each positive sample.
- **dropout_rate** (*float, default: 0.0*) – Probability of a node being dropped. 0.0 means dropout is not used.
- **remove_edges** (*bool, default: False*) – Whether to remove edges between target node and its positive pair nodes when target node's sampled neighbor nodes contain positive pair nodes. This only applies in 'i2i' paradigm.
- **num_layers** (*int, default: 2*) – Number of GCN layers.
- **num_neighbors** (*int, default: 3*) – Number of sampled neighbors in each layer
- **num_walks** (*int, default: 10*) – Number of random walks to sample positive item pairs. This only applies in 'i2i' paradigm.
- **sample_walk_len** (*int, default: 5*) – Length of each random walk to sample positive item pairs.
- **margin** (*float, default: 1.0*) – Margin used in *max_margin* loss.

- **sampler** (*{'random', 'unconsumed', 'popular', 'out-batch'}, default: 'random'*) – Negative sampling strategy. The `'u2i'` paradigm can use `'random'`, `'unconsumed'`, `'popular'`, and the `'i2i'` paradigm can use `'random'`, `'out-batch'`, `'popular'`.
  - `'random'` means random sampling.
  - `'unconsumed'` samples items that the target user did not consume before. This can't be used in `'i2i'` since it has no users.
  - `'popular'` has a higher probability to sample popular items as negative samples.
  - `'out-batch'` samples items that didn't appear in the batch. This can only be used in `'i2i'` paradigm.
- **start_node** (*{'random', 'unpopular'}, default: 'random'*) – Strategy for choosing start nodes in random walks. `'unpopular'` will place a higher probability on unpopular items, which may increase diversity but hurt metrics. This only applies in `'i2i'` paradigm.
- **focus_start** (*bool, default: False*) – Whether to keep the start nodes in random walk sampling. The purpose of the parameter `start_node` and `focus_start` is oversampling unpopular items. If you set `start_node='popular'` and `focus_start=True`, unpopular items will be kept in positive samples, which may increase diversity.
- **seed** (*int, default: 42*) – Random seed.
- **device** (*{'cpu', 'cuda'}, default: 'cuda'*) – Refer to torch.device.

  Changed in version 1.0.0: Accept str type `'cpu'` or `'cuda'`, instead of `torch.device(..)`.
- **lower_upper_bound** (*tuple or None, default: None*) – Lower and upper score bound for *rating* task.

See also:

GraphSageDGL

References

William L. Hamilton et al. Inductive Representation Learning on Large Graphs.

fit(*train_data*, *verbose=1*, *shuffle=True*, *eval_data=None*, *metrics=None*, *k=10*, *eval_batch_size=8192*, *eval_user_num=None*)

Fit embed model on the training data.

Parameters

- **train_data** (*TransformedSet* object) – Data object used for training.
- **verbose** (*int, default: 1*) – Print verbosity. If *eval_data* is provided, setting it to higher than 1 will print evaluation metrics during training.
- **shuffle** (*bool, default: True*) – Whether to shuffle the training data.
- **eval_data** (*TransformedSet* object, default: None) – Data object used for evaluating.
- **metrics** (*list or None, default: None*) – List of metrics for evaluating.
- **k** (*int, default: 10*) – Parameter of metrics, e.g. recall at k, ndcg at k
- **eval_batch_size** (*int, default: 8192*) – Batch size for evaluating.

- **eval_user_num** (*int or None, default: None*) – Number of users for evaluating. Setting it to a positive number will sample users randomly from eval data.

Raises

RuntimeError – If *fit()* is called from a loaded model(*load()*).

get_item_embedding(*item=None*)

Get item embedding(s) from the model.

Parameters

item (*int or str or None*) – Query item id. If it is None, all item embeddings will be returned.

Returns

item_embedding – Returned item embeddings.

Return type

numpy.ndarray

Raises

- **ValueError** – If the item does not appear in the training data.
- **AssertionError** – If the model has not been trained.

get_user_embedding(*user=None*)

Get user embedding(s) from the model.

Parameters

user (*int or str or None*) – Query user id. If it is None, all user embeddings will be returned.

Returns

user_embedding – Returned user embeddings.

Return type

numpy.ndarray

Raises

- **ValueError** – If the user does not appear in the training data.
- **AssertionError** – If the model has not been trained.

init_knn(*approximate*, *sim_type*, *M=100*, *ef_construction=200*, *ef_search=200*)

Initialize k-nearest-search model.

Parameters

- **approximate** (*bool*) – Whether to use approximate nearest neighbor search. If it is True, nmslib must be installed. The *HNSW* method in *nmslib* is used.
- **sim_type** (*{'cosine', 'inner-product'}*) – Similarity space type.
- **M** (*int, default: 100*) – Parameter in *HNSW*, refer to nmslib doc.
- **ef_construction** (*int, default: 200*) – Parameter in *HNSW*, refer to nmslib doc.
- **ef_search** (*int, default: 200*) – Parameter in *HNSW*, refer to nmslib doc.

Raises

- **ValueError** – If sim_type is not one of ('cosine', 'inner-product').
- **ModuleNotFoundError** – If *approximate=True* and *nmslib* is not installed.

classmethod load(*path*, *model_name*, *data_info*, ***kwargs*)

Load saved embed model for inference.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.
- **data_info** (*DataInfo* object) – Object that contains some useful information.

Returns
model – Loaded embed model.

Return type
type(cls)

See also:

save

predict(*user*, *item*, *cold_start='average'*, *inner_id=False*)

Make prediction(s) on given user(s) and item(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids.
- **item** (*int or str or array_like*) – Item id or batch of item ids.
- **cold_start** (*{'popular', 'average'}, default: 'average'*) – Cold start strategy.
  - 'popular' will sample from popular items.
  - 'average' will use the average of all the user/item embeddings as the representation of the cold-start user/item.
- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.

Returns
prediction – Predicted scores for each user-item pair.

Return type
float or numpy.ndarray

rebuild_model(*path*, *model_name*)

Assign the saved model variables to the newly initialized model.

This method is used before retraining the new model, in order to avoid training from scratch every time we get some new data.

Parameters

- **path** (*str*) – File folder path for the saved model variables.
- **model_name** (*str*) – Name of the saved model file.

recommend_user(*user*, *n_rec*, *cold_start='average'*, *inner_id=False*, *filter_consumed=True*, *random_rec=False*)

Recommend a list of items for given user(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids to recommend.

- **n_rec** (*int*) – Number of recommendations to return.
- **cold_start** (*{'popular', 'average'}, default: 'average'*) – Cold start strategy.
  - 'popular' will sample from popular items.
  - 'average' will use the average of all the user/item embeddings as the representation of the cold-start user/item.
- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.
- **filter_consumed** (*bool, default: True*) – Whether to filter out items that a user has previously consumed.
- **random_rec** (*bool, default: False*) – Whether to choose items for recommendation based on their prediction scores.

Returns
recommendation – Recommendation result with user ids as keys and array_like recommended items as values.

Return type
dict of {Union[int, str, array_like] : numpy.ndarray}

save(*path*, *model_name*, *inference_only=False*, ***kwargs*)

Save embed model for inference or retraining.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.
- **inference_only** (*bool, default: False*) – Whether to save model only for inference. If it is True, only embeddings will be saved. Otherwise, model variables will be saved.

See also:

load

search_knn_items(*item*, *k*)

Search most similar k items.

Parameters

- **item** (*int or str*) – Query item id.
- **k** (*int*) – Number of similar items.

Returns
similar items – A list of k similar items.

Return type
list

search_knn_users(*user*, *k*)

Search most similar k users.

Parameters

- **user** (*int or str*) – Query user id.
- **k** (*int*) – Number of similar users.

Returns
similar users – A list of k similar users.

Return type
list

1.14.24 GraphSageDGL

class libreco.algorithms.**GraphSageDGL**(**args*, ***kwargs*)

Bases: *EmbedBase*

GraphSageDGL algorithm.

Note: This algorithm is implemented in DGL.

Caution: GraphSageDGL can only be used in `ranking` task.

New in version 0.12.0.

Parameters

- **task** (*{'ranking'}*) – Recommendation task. See *Task*.
- **data_info** (*DataInfo* object) – Object that contains useful information for training and inference.
- **loss_type** (*{'cross_entropy', 'focal', 'bpr', 'max_margin'}, default: 'cross_entropy'*) – Loss for model training.
- **paradigm** (*{'u2i', 'i2i'}, default: 'i2i'*) – Choice for features in model.
  - `'u2i'` will combine user features and item features.
  - `'i2i'` will only use item features, this is the setting in the original paper.
- **aggregator_type** (*{'mean', 'gcn', 'pool', 'lstm'}, default: 'mean'*) – Aggregator type to use in GraphSage. Refer to SAGEConv in DGL.
- **embed_size** (*int, default: 16*) – Vector size of embeddings.
- **n_epochs** (*int, default: 10*) – Number of epochs for training.
- **lr** (*float, default 0.001*) – Learning rate for training.
- **lr_decay** (*bool, default: False*) – Whether to use learning rate decay.
- **epsilon** (*float, default: 1e-8*) – A small constant added to the denominator to improve numerical stability in Adam optimizer.
- **amsgrad** (*bool, default: False*) – Whether to use the AMSGrad variant from the paper On the Convergence of Adam and Beyond.
- **reg** (*float or None, default: None*) – Regularization parameter, must be non-negative or None.
- **batch_size** (*int, default: 256*) – Batch size for training.
- **num_neg** (*int, default: 1*) – Number of negative samples for each positive sample.

- **dropout_rate** (*float, default: 0.0*) – Probability of a node being dropped. 0.0 means dropout is not used.
- **remove_edges** (*bool, default: False*) – Whether to remove edges between target node and its positive pair nodes when target node's sampled neighbor nodes contain positive pair nodes. This only applies in 'i2i' paradigm.
- **num_layers** (*int, default: 2*) – Number of GCN layers.
- **num_neighbors** (*int, default: 3*) – Number of sampled neighbors in each layer
- **num_walks** (*int, default: 10*) – Number of random walks to sample positive item pairs. This only applies in 'i2i' paradigm.
- **sample_walk_len** (*int, default: 5*) – Length of each random walk to sample positive item pairs.
- **margin** (*float, default: 1.0*) – Margin used in *max_margin* loss.
- **sampler** (*{'random', 'unconsumed', 'popular', 'out-batch'}, default: 'random'*) – Negative sampling strategy. The `'u2i'` paradigm can use `'random'`, `'unconsumed'`, `'popular'`, and the `'i2i'` paradigm can use `'random'`, `'out-batch'`, `'popular'`.
  - `'random'` means random sampling.
  - `'unconsumed'` samples items that the target user did not consume before. This can't be used in `'i2i'` since it has no users.
  - `'popular'` has a higher probability to sample popular items as negative samples.
  - `'out-batch'` samples items that didn't appear in the batch. This can only be used in `'i2i'` paradigm.
- **start_node** (*{'random', 'unpopular'}, default: 'random'*) – Strategy for choosing start nodes in random walks. `'unpopular'` will place a higher probability on unpopular items, which may increase diversity but hurt metrics. This only applies in `'i2i'` paradigm.
- **focus_start** (*bool, default: False*) – Whether to keep the start nodes in random walk sampling. The purpose of the parameter `start_node` and `focus_start` is oversampling unpopular items. If you set `start_node='popular'` and `focus_start=True`, unpopular items will be kept in positive samples, which may increase diversity.
- **seed** (*int, default: 42*) – Random seed.
- **device** (*{'cpu', 'cuda'}, default: 'cuda'*) – Refer to torch.device.

  Changed in version 1.0.0: Accept str type `'cpu'` or `'cuda'`, instead of `torch.device(...)`.
- **lower_upper_bound** (*tuple or None, default: None*) – Lower and upper score bound for *rating* task.

See also:

GraphSage

References

William L. Hamilton et al. Inductive Representation Learning on Large Graphs.

fit(*train_data*, *verbose=1*, *shuffle=True*, *eval_data=None*, *metrics=None*, *k=10*, *eval_batch_size=8192*, *eval_user_num=None*)

Fit embed model on the training data.

Parameters

- **train_data** (*TransformedSet* object) – Data object used for training.
- **verbose** (*int, default: 1*) – Print verbosity. If *eval_data* is provided, setting it to higher than 1 will print evaluation metrics during training.
- **shuffle** (*bool, default: True*) – Whether to shuffle the training data.
- **eval_data** (*TransformedSet* object, default: None) – Data object used for evaluating.
- **metrics** (*list or None, default: None*) – List of metrics for evaluating.
- **k** (*int, default: 10*) – Parameter of metrics, e.g. recall at k, ndcg at k
- **eval_batch_size** (*int, default: 8192*) – Batch size for evaluating.
- **eval_user_num** (*int or None, default: None*) – Number of users for evaluating. Setting it to a positive number will sample users randomly from eval data.

Raises

RuntimeError – If *fit()* is called from a loaded model(*load()*).

get_item_embedding(*item=None*)

Get item embedding(s) from the model.

Parameters

item (*int or str or None*) – Query item id. If it is None, all item embeddings will be returned.

Returns

item_embedding – Returned item embeddings.

Return type

numpy.ndarray

Raises

- **ValueError** – If the item does not appear in the training data.
- **AssertionError** – If the model has not been trained.

get_user_embedding(*user=None*)

Get user embedding(s) from the model.

Parameters

user (*int or str or None*) – Query user id. If it is None, all user embeddings will be returned.

Returns

user_embedding – Returned user embeddings.

Return type

numpy.ndarray

Raises

- **ValueError** – If the user does not appear in the training data.
- **AssertionError** – If the model has not been trained.

init_knn(*approximate*, *sim_type*, *M=100*, *ef_construction=200*, *ef_search=200*)

Initialize k-nearest-search model.

Parameters

- **approximate** (*bool*) – Whether to use approximate nearest neighbor search. If it is True, nmslib must be installed. The *HNSW* method in *nmslib* is used.
- **sim_type** (*{'cosine', 'inner-product'}*) – Similarity space type.
- **M** (*int, default: 100*) – Parameter in *HNSW*, refer to nmslib doc.
- **ef_construction** (*int, default: 200*) – Parameter in *HNSW*, refer to nmslib doc.
- **ef_search** (*int, default: 200*) – Parameter in *HNSW*, refer to nmslib doc.

Raises

- **ValueError** – If sim_type is not one of ('cosine', 'inner-product').
- **ModuleNotFoundError** – If *approximate=True* and *nmslib* is not installed.

classmethod load(*path*, *model_name*, *data_info*, ***kwargs*)

Load saved embed model for inference.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.
- **data_info** (*DataInfo* object) – Object that contains some useful information.

Returns

model – Loaded embed model.

Return type

type(cls)

See also:

save

predict(*user*, *item*, *cold_start='average'*, *inner_id=False*)

Make prediction(s) on given user(s) and item(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids.
- **item** (*int or str or array_like*) – Item id or batch of item ids.
- **cold_start** (*{'popular', 'average'}, default: 'average'*) – Cold start strategy.
  - 'popular' will sample from popular items.
  - 'average' will use the average of all the user/item embeddings as the representation of the cold-start user/item.
- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.

Returns

prediction – Predicted scores for each user-item pair.

Return type

float or numpy.ndarray

rebuild_model(*path*, *model_name*)

Assign the saved model variables to the newly initialized model.

This method is used before retraining the new model, in order to avoid training from scratch every time we get some new data.

Parameters

- **path** (*str*) – File folder path for the saved model variables.
- **model_name** (*str*) – Name of the saved model file.

recommend_user(*user*, *n_rec*, *cold_start='average'*, *inner_id=False*, *filter_consumed=True*, *random_rec=False*)

Recommend a list of items for given user(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids to recommend.
- **n_rec** (*int*) – Number of recommendations to return.
- **cold_start** (*{'popular', 'average'}, default: 'average'*) – Cold start strategy.
  - 'popular' will sample from popular items.
  - 'average' will use the average of all the user/item embeddings as the representation of the cold-start user/item.
- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.
- **filter_consumed** (*bool, default: True*) – Whether to filter out items that a user has previously consumed.
- **random_rec** (*bool, default: False*) – Whether to choose items for recommendation based on their prediction scores.

Returns

recommendation – Recommendation result with user ids as keys and array_like recommended items as values.

Return type

dict of {Union[int, str, array_like] : numpy.ndarray}

save(*path*, *model_name*, *inference_only=False*, ***kwargs*)

Save embed model for inference or retraining.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.
- **inference_only** (*bool, default: False*) – Whether to save model only for inference. If it is True, only embeddings will be saved. Otherwise, model variables will be saved.

See also:

load

search_knn_items(*item*, *k*)

Search most similar k items.

Parameters

- **item** (*int or str*) – Query item id.
- **k** (*int*) – Number of similar items.

Returns
similar items – A list of k similar items.

Return type
list

`search_knn_users`(*user*, *k*)

Search most similar k users.

Parameters

- **user** (*int or str*) – Query user id.
- **k** (*int*) – Number of similar users.

Returns
similar users – A list of k similar users.

Return type
list

1.14.25 PinSage

`class libreco.algorithms.PinSage`(*task*, *data_info*, *loss_type='max_margin'*, *paradigm='i2i'*, *embed_size=16*, *n_epochs=20*, *lr=0.001*, *lr_decay=False*, *epsilon=1e-08*, *amsgrad=False*, *reg=None*, *batch_size=256*, *num_neg=1*, *dropout_rate=0.0*, *remove_edges=False*, *num_layers=2*, *num_neighbors=3*, *num_walks=10*, *neighbor_walk_len=2*, *sample_walk_len=5*, *termination_prob=0.5*, *margin=1.0*, *sampler='random'*, *start_node='random'*, *focus_start=False*, *seed=42*, *device='cuda'*, *lower_upper_bound=None*)

Bases: *GraphSage*

PinSage algorithm.

Note: This algorithm is implemented in PyTorch.

Caution: PinSage can only be used in `ranking` task.

New in version 0.12.0.

Parameters

- **task** (*{'ranking'}*) – Recommendation task. See *Task*.
- **data_info** (*DataInfo* object) – Object that contains useful information for training and inference.
- **loss_type** (*{'cross_entropy', 'focal', 'bpr', 'max_margin'}, default: 'max_margin'*) – Loss for model training.
- **paradigm** (*{'u2i', 'i2i'}, default: 'i2i'*) – Choice for features in model.

  - `'u2i'` will combine user features and item features.
  - `'i2i'` will only use item features, this is the setting in the original paper.
- **embed_size** (*int, default: 16*) – Vector size of embeddings.
- **n_epochs** (*int, default: 10*) – Number of epochs for training.
- **lr** (*float, default 0.001*) – Learning rate for training.
- **lr_decay** (*bool, default: False*) – Whether to use learning rate decay.
- **epsilon** (*float, default: 1e-8*) – A small constant added to the denominator to improve numerical stability in Adam optimizer.
- **amsgrad** (*bool, default: False*) – Whether to use the AMSGrad variant from the paper On the Convergence of Adam and Beyond.
- **reg** (*float or None, default: None*) – Regularization parameter, must be non-negative or None.
- **batch_size** (*int, default: 256*) – Batch size for training.
- **num_neg** (*int, default: 1*) – Number of negative samples for each positive sample.
- **dropout_rate** (*float, default: 0.0*) – Probability of a node being dropped. 0.0 means dropout is not used.
- **remove_edges** (*bool, default: False*) – Whether to remove edges between target node and its positive pair nodes when target node's sampled neighbor nodes contain positive pair nodes. This only applies in 'i2i' paradigm.
- **num_layers** (*int, default: 2*) – Number of GCN layers.
- **num_neighbors** (*int, default: 3*) – Number of sampled neighbors in each layer
- **num_walks** (*int, default: 10*) – Number of random walks to sample positive item pairs. This only applies in 'i2i' paradigm.
- **neighbor_walk_len** (*int, default: 2*) – Length of random walk to sample neighbor nodes for target node.
- **sample_walk_len** (*int, default: 5*) – Length of each random walk to sample positive item pairs.
- **termination_prob** (*float, default: 0.5*) – Termination probability after one walk for neighbor random walk sampling.
- **margin** (*float, default: 1.0*) – Margin used in *max_margin* loss.
- **sampler** (*{'random', 'unconsumed', 'popular', 'out-batch'}, default: 'random'*) – Negative sampling strategy. The `'u2i'` paradigm can use `'random'`, `'unconsumed'`, `'popular'`, and the `'i2i'` paradigm can use `'random'`, `'out-batch'`, `'popular'`.
  - `'random'` means random sampling.
  - `'unconsumed'` samples items that the target user did not consume before. This can't be used in `'i2i'` since it has no users.
  - `'popular'` has a higher probability to sample popular items as negative samples.
  - `'out-batch'` samples items that didn't appear in the batch. This can only be used in `'i2i'` paradigm.

- **start_node** (*{'random', 'unpopular'}, default: 'random'*) – Strategy for choosing start nodes in random walks. `'unpopular'` will place a higher probability on unpopular items, which may increase diversity but hurt metrics. This only applies in `'i2i'` paradigm.
- **focus_start** (*bool, default: False*) – Whether to keep the start nodes in random walk sampling. The purpose of the parameter `start_node` and `focus_start` is oversampling unpopular items. If you set `start_node='popular'` and `focus_start=True`, unpopular items will be kept in positive samples, which may increase diversity.
- **seed** (*int, default: 42*) – Random seed.
- **device** (*{'cpu', 'cuda'}, default: 'cuda'*) – Refer to torch.device.

  Changed in version 1.0.0: Accept str type `'cpu'` or `'cuda'`, instead of `torch.device(..)`.
- **lower_upper_bound** (*tuple or None, default: None*) – Lower and upper score bound for *rating* task.

See also:

PinSageDGL

References

Rex Ying et al. Graph Convolutional Neural Networks for Web-Scale Recommender Systems.

---

fit(*train_data*, *verbose=1*, *shuffle=True*, *eval_data=None*, *metrics=None*, *k=10*, *eval_batch_size=8192*, *eval_user_num=None*)

Fit embed model on the training data.

Parameters

- **train_data** (*TransformedSet* object) – Data object used for training.
- **verbose** (*int, default: 1*) – Print verbosity. If *eval_data* is provided, setting it to higher than 1 will print evaluation metrics during training.
- **shuffle** (*bool, default: True*) – Whether to shuffle the training data.
- **eval_data** (*TransformedSet* object, default: None) – Data object used for evaluating.
- **metrics** (*list or None, default: None*) – List of metrics for evaluating.
- **k** (*int, default: 10*) – Parameter of metrics, e.g. recall at k, ndcg at k
- **eval_batch_size** (*int, default: 8192*) – Batch size for evaluating.
- **eval_user_num** (*int or None, default: None*) – Number of users for evaluating. Setting it to a positive number will sample users randomly from eval data.

Raises

RuntimeError – If *fit()* is called from a loaded model(*load()*).

get_item_embedding(*item=None*)

Get item embedding(s) from the model.

Parameters

item (*int or str or None*) – Query item id. If it is None, all item embeddings will be returned.

Returns

item_embedding – Returned item embeddings.

Return type
numpy.ndarray

Raises

- **ValueError** – If the item does not appear in the training data.
- **AssertionError** – If the model has not been trained.

get_user_embedding(*user=None*)

Get user embedding(s) from the model.

Parameters
user (*int or str or None*) – Query user id. If it is None, all user embeddings will be returned.

Returns
user_embedding – Returned user embeddings.

Return type
numpy.ndarray

Raises

- **ValueError** – If the user does not appear in the training data.
- **AssertionError** – If the model has not been trained.

init_knn(*approximate*, *sim_type*, *M=100*, *ef_construction=200*, *ef_search=200*)

Initialize k-nearest-search model.

Parameters

- **approximate** (*bool*) – Whether to use approximate nearest neighbor search. If it is True, nmslib must be installed. The *HNSW* method in *nmslib* is used.
- **sim_type** (*{'cosine', 'inner-product'}*) – Similarity space type.
- **M** (*int, default: 100*) – Parameter in *HNSW*, refer to nmslib doc.
- **ef_construction** (*int, default: 200*) – Parameter in *HNSW*, refer to nmslib doc.
- **ef_search** (*int, default: 200*) – Parameter in *HNSW*, refer to nmslib doc.

Raises

- **ValueError** – If sim_type is not one of ('cosine', 'inner-product').
- **ModuleNotFoundError** – If *approximate=True* and *nmslib* is not installed.

classmethod load(*path*, *model_name*, *data_info*, ***kwargs*)

Load saved embed model for inference.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.
- **data_info** (*DataInfo* object) – Object that contains some useful information.

Returns
model – Loaded embed model.

Return type
type(cls)

See also:

save

predict(*user*, *item*, *cold_start='average'*, *inner_id=False*)

Make prediction(s) on given user(s) and item(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids.
- **item** (*int or str or array_like*) – Item id or batch of item ids.
- **cold_start** (*{'popular', 'average'}, default: 'average'*) – Cold start strategy.
  - 'popular' will sample from popular items.
  - 'average' will use the average of all the user/item embeddings as the representation of the cold-start user/item.
- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.

Returns

prediction – Predicted scores for each user-item pair.

Return type

float or numpy.ndarray

rebuild_model(*path*, *model_name*)

Assign the saved model variables to the newly initialized model.

This method is used before retraining the new model, in order to avoid training from scratch every time we get some new data.

Parameters

- **path** (*str*) – File folder path for the saved model variables.
- **model_name** (*str*) – Name of the saved model file.

recommend_user(*user*, *n_rec*, *cold_start='average'*, *inner_id=False*, *filter_consumed=True*, *random_rec=False*)

Recommend a list of items for given user(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids to recommend.
- **n_rec** (*int*) – Number of recommendations to return.
- **cold_start** (*{'popular', 'average'}, default: 'average'*) – Cold start strategy.
  - 'popular' will sample from popular items.
  - 'average' will use the average of all the user/item embeddings as the representation of the cold-start user/item.
- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.
- **filter_consumed** (*bool, default: True*) – Whether to filter out items that a user has previously consumed.
- **random_rec** (*bool, default: False*) – Whether to choose items for recommendation based on their prediction scores.

Returns
recommendation – Recommendation result with user ids as keys and array_like recommended items as values.

Return type
dict of {Union[int, str, array_like] : numpy.ndarray}

`save`(*path*, *model_name*, *inference_only=False*, ***kwargs*)

Save embed model for inference or retraining.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.
- **inference_only** (*bool, default: False*) – Whether to save model only for inference. If it is True, only embeddings will be saved. Otherwise, model variables will be saved.

See also:

load

`search_knn_items`(*item*, *k*)

Search most similar k items.

Parameters

- **item** (*int or str*) – Query item id.
- **k** (*int*) – Number of similar items.

Returns
similar items – A list of k similar items.

Return type
list

`search_knn_users`(*user*, *k*)

Search most similar k users.

Parameters

- **user** (*int or str*) – Query user id.
- **k** (*int*) – Number of similar users.

Returns
similar users – A list of k similar users.

Return type
list

1.14.26 PinSageDGL

class libreco.algorithms.**PinSageDGL**(**args*, ***kwargs*)

Bases: *GraphSageDGL*

PinSageDGL algorithm.

Note: This algorithm is implemented in DGL.

Caution: PinSageDGL can only be used in `ranking` task.

New in version 0.12.0.

Parameters

- **task** (*{'ranking'}*) – Recommendation task. See *Task*.
- **data_info** (*DataInfo* object) – Object that contains useful information for training and inference.
- **loss_type** (*{'cross_entropy', 'focal', 'bpr', 'max_margin'}, default: 'max_margin'*) – Loss for model training.
- **paradigm** (*{'u2i', 'i2i'}, default: 'i2i'*) – Choice for features in model.
  - `'u2i'` will combine user features and item features.
  - `'i2i'` will only use item features, this is the setting in the original paper.
- **embed_size** (*int, default: 16*) – Vector size of embeddings.
- **n_epochs** (*int, default: 10*) – Number of epochs for training.
- **lr** (*float, default 0.001*) – Learning rate for training.
- **lr_decay** (*bool, default: False*) – Whether to use learning rate decay.
- **epsilon** (*float, default: 1e-8*) – A small constant added to the denominator to improve numerical stability in Adam optimizer.
- **amsgrad** (*bool, default: False*) – Whether to use the AMSGrad variant from the paper On the Convergence of Adam and Beyond.
- **reg** (*float or None, default: None*) – Regularization parameter, must be non-negative or None.
- **batch_size** (*int, default: 256*) – Batch size for training.
- **num_neg** (*int, default: 1*) – Number of negative samples for each positive sample.
- **dropout_rate** (*float, default: 0.0*) – Probability of a node being dropped. 0.0 means dropout is not used.
- **remove_edges** (*bool, default: False*) – Whether to remove edges between target node and its positive pair nodes when target node's sampled neighbor nodes contain positive pair nodes. This only applies in 'i2i' paradigm.
- **num_layers** (*int, default: 2*) – Number of GCN layers.
- **num_neighbors** (*int, default: 3*) – Number of sampled neighbors in each layer

- **num_walks** (*int, default: 10*) – Number of random walks to sample positive item pairs. This only applies in 'i2i' paradigm.
- **neighbor_walk_len** (*int, default: 2*) – Length of random walk to sample neighbor nodes for target node.
- **sample_walk_len** (*int, default: 5*) – Length of each random walk to sample positive item pairs.
- **termination_prob** (*float, default: 0.5*) – Termination probability after one walk for neighbor random walk sampling.
- **margin** (*float, default: 1.0*) – Margin used in *max_margin* loss.
- **sampler** (*{'random', 'unconsumed', 'popular', 'out-batch'}, default: 'random'*) – Negative sampling strategy. The `'u2i'` paradigm can use `'random'`, `'unconsumed'`, `'popular'`, and the `'i2i'` paradigm can use `'random'`, `'out-batch'`, `'popular'`.
  - `'random'` means random sampling.
  - `'unconsumed'` samples items that the target user did not consume before. This can't be used in `'i2i'` since it has no users.
  - `'popular'` has a higher probability to sample popular items as negative samples.
  - `'out-batch'` samples items that didn't appear in the batch. This can only be used in `'i2i'` paradigm.
- **start_node** (*{'random', 'unpopular'}, default: 'random'*) – Strategy for choosing start nodes in random walks. `'unpopular'` will place a higher probability on unpopular items, which may increase diversity but hurt metrics. This only applies in `'i2i'` paradigm.
- **focus_start** (*bool, default: False*) – Whether to keep the start nodes in random walk sampling. The purpose of the parameter `start_node` and `focus_start` is oversampling unpopular items. If you set `start_node='popular'` and `focus_start=True`, unpopular items will be kept in positive samples, which may increase diversity.
- **seed** (*int, default: 42*) – Random seed.
- **device** (*{'cpu', 'cuda'}, default: 'cuda'*) – Refer to torch.device.

  Changed in version 1.0.0: Accept str type `'cpu'` or `'cuda'`, instead of `torch.device(..)`.
- **lower_upper_bound** (*tuple or None, default: None*) – Lower and upper score bound for *rating* task.

See also:

PinSage

References

Rex Ying et al. Graph Convolutional Neural Networks for Web-Scale Recommender Systems.

fit(*train_data*, *verbose=1*, *shuffle=True*, *eval_data=None*, *metrics=None*, *k=10*, *eval_batch_size=8192*, *eval_user_num=None*)

Fit embed model on the training data.

Parameters

- **train_data** (*TransformedSet* object) – Data object used for training.

- **verbose** (*int, default: 1*) – Print verbosity. If *eval_data* is provided, setting it to higher than 1 will print evaluation metrics during training.
- **shuffle** (*bool, default: True*) – Whether to shuffle the training data.
- **eval_data** (*TransformedSet* object, default: None) – Data object used for evaluating.
- **metrics** (*list or None, default: None*) – List of metrics for evaluating.
- **k** (*int, default: 10*) – Parameter of metrics, e.g. recall at k, ndcg at k
- **eval_batch_size** (*int, default: 8192*) – Batch size for evaluating.
- **eval_user_num** (*int or None, default: None*) – Number of users for evaluating. Setting it to a positive number will sample users randomly from eval data.

Raises
RuntimeError – If *fit()* is called from a loaded model(*load()*).

get_item_embedding(*item=None*)

Get item embedding(s) from the model.

Parameters
item (*int or str or None*) – Query item id. If it is None, all item embeddings will be returned.

Returns
item_embedding – Returned item embeddings.

Return type
numpy.ndarray

Raises
- **ValueError** – If the item does not appear in the training data.
- **AssertionError** – If the model has not been trained.

get_user_embedding(*user=None*)

Get user embedding(s) from the model.

Parameters
user (*int or str or None*) – Query user id. If it is None, all user embeddings will be returned.

Returns
user_embedding – Returned user embeddings.

Return type
numpy.ndarray

Raises
- **ValueError** – If the user does not appear in the training data.
- **AssertionError** – If the model has not been trained.

init_knn(*approximate*, *sim_type*, *M=100*, *ef_construction=200*, *ef_search=200*)

Initialize k-nearest-search model.

Parameters
- **approximate** (*bool*) – Whether to use approximate nearest neighbor search. If it is True, nmslib must be installed. The *HNSW* method in *nmslib* is used.
- **sim_type** (*{'cosine', 'inner-product'}*) – Similarity space type.

- **M** (*int, default: 100*) – Parameter in *HNSW*, refer to nmslib doc.
- **ef_construction** (*int, default: 200*) – Parameter in *HNSW*, refer to nmslib doc.
- **ef_search** (*int, default: 200*) – Parameter in *HNSW*, refer to nmslib doc.

Raises

- **ValueError** – If sim_type is not one of ('cosine', 'inner-product').
- **ModuleNotFoundError** – If *approximate=True* and *nmslib* is not installed.

classmethod load(*path*, *model_name*, *data_info*, ***kwargs*)

Load saved embed model for inference.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.
- **data_info** (*DataInfo* object) – Object that contains some useful information.

Returns
model – Loaded embed model.

Return type
type(cls)

See also:

save

predict(*user*, *item*, *cold_start='average'*, *inner_id=False*)

Make prediction(s) on given user(s) and item(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids.
- **item** (*int or str or array_like*) – Item id or batch of item ids.
- **cold_start** (*{'popular', 'average'}, default: 'average'*) – Cold start strategy.
  - 'popular' will sample from popular items.
  - 'average' will use the average of all the user/item embeddings as the representation of the cold-start user/item.
- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.

Returns
prediction – Predicted scores for each user-item pair.

Return type
float or numpy.ndarray

rebuild_model(*path*, *model_name*)

Assign the saved model variables to the newly initialized model.

This method is used before retraining the new model, in order to avoid training from scratch every time we get some new data.

Parameters

- **path** (*str*) – File folder path for the saved model variables.

- **model_name** (*str*) – Name of the saved model file.

recommend_user(*user*, *n_rec*, *cold_start='average'*, *inner_id=False*, *filter_consumed=True*, *random_rec=False*)

Recommend a list of items for given user(s).

Parameters

- **user** (*int or str or array_like*) – User id or batch of user ids to recommend.
- **n_rec** (*int*) – Number of recommendations to return.
- **cold_start** (*{'popular', 'average'}, default: 'average'*) – Cold start strategy.
  - 'popular' will sample from popular items.
  - 'average' will use the average of all the user/item embeddings as the representation of the cold-start user/item.
- **inner_id** (*bool, default: False*) – Whether to use inner_id defined in *libreco*. For library users inner_id may never be used.
- **filter_consumed** (*bool, default: True*) – Whether to filter out items that a user has previously consumed.
- **random_rec** (*bool, default: False*) – Whether to choose items for recommendation based on their prediction scores.

Returns

recommendation – Recommendation result with user ids as keys and array_like recommended items as values.

Return type

dict of {Union[int, str, array_like] : numpy.ndarray}

save(*path*, *model_name*, *inference_only=False*, ***kwargs*)

Save embed model for inference or retraining.

Parameters

- **path** (*str*) – File folder path to save model.
- **model_name** (*str*) – Name of the saved model file.
- **inference_only** (*bool, default: False*) – Whether to save model only for inference. If it is True, only embeddings will be saved. Otherwise, model variables will be saved.

See also:

load

search_knn_items(*item*, *k*)

Search most similar k items.

Parameters

- **item** (*int or str*) – Query item id.
- **k** (*int*) – Number of similar items.

Returns

similar items – A list of k similar items.

Return type

list

search_knn_users(*user*, *k*)

Search most similar k users.

Parameters

- **user** (*int or str*) – Query user id.
- **k** (*int*) – Number of similar users.

Returns

similar users – A list of k similar users.

Return type

list

1.15 Evaluation

libreco.evaluation.**evaluate**(*model*, *data*, *eval_batch_size=8192*, *metrics=None*, *k=10*, *sample_user_num=2048*, *neg_sample=False*, *update_features=False*, *seed=42*)

Evaluate the model on specific data and metrics.

Parameters

- **model** (Base) – Model for evaluation.
- **data** (pandas.DataFrame or *TransformedSet*) – Data to evaluate.
- **eval_batch_size** (*int, default: 8192*) – Batch size used in evaluation.
- **metrics** (*list or None, default: None*) – List of metrics for evaluating.
- **k** (*int, default: 10*) – Parameter of metrics, e.g. recall at k, ndcg at k
- **sample_user_num** (*int, default: 2048*) – Number of users for evaluating. Setting it to a positive number will sample users randomly from eval data.
- **neg_sample** (*bool, default: False*) – Whether to do negative sampling when evaluating.
- **update_features** (*bool, default: False*) – Whether to update model's data_info from features in data.
- **seed** (*int, default: 42*) – Random seed.

Returns

results – Evaluation results for the model and data.

Return type

dict of {str : float}

Examples

```
>>> eval_result = evaluate(model, data, metrics=["roc_auc", "precision", "recall"])
```

1.16 Indices and tables

- genindex

PYTHON MODULE INDEX

INDEX

Symbols

A

B

C

D

E

F

G

I

L

M

N

P

R

S

T

U

W

Y