Recoder Documentation

Release 0.4.0

Abdallah Moussawi

Apr 11, 2021

Contents:

CHAPTER 1

Recoder

Pypi version Docs status Build Status

1.1 Introduction

Recoder is a fast implementation for training collaborative filtering latent factor models with mini-batch based negative sampling following recent work:

- Towards Large Scale Training Of Autoencoders For Collaborative Filtering.

Recoder contains two implementations of factorization models: Autoencoder and Matrix Factorization.

Check out the Documentation and the Tutorial.

1.2 Installation

Recommended to use python 3.6. Python 2 is not supported.

```
pip install -U recsys-recoder
```

1.3 Examples

Check out the `scripts/` directory for some good examples on different datasets. You can get MovieLens-20M dataset fully trained with mean squared error in less than a minute on a Nvidia Tesla K80 GPU.

1.4 Further Readings

- Collaborative Filtering for Implicit Feedback Datasets

- Variational Autoencoders for Collaborative Filtering

1.5 Citing

Please cite this paper in your publications if it helps your research:

```
@inproceedings{recoder,
  author = {Moussawi, Abdallah},
  title = {Towards Large Scale Training Of Autoencoders For Collaborative Filtering},
  booktitle = {Proceedings of Late-Breaking Results track part of the Twelfth ACM␣
↪Conference on Recommender Systems},
  series = {RecSys'18},
  year = {2018},
  address = {Vancouver, BC, Canada}
}
```

1.6 Acknowledgements

- I would like to thank Anghami for supporting this work, and my colleagues, Helmi Rifai and Ramzi Karam, for great discussions on Collaborative Filtering at scale.
- This project started as a fork of NVIDIA/DeepRecommender, and although it went in a slightly different direction and was entirely refactored, the work in NVIDIA/DeepRecommender was a great contribution to the work here.

CHAPTER 2

Tutorial

In this quick tutorial, we will show how to:

- Train an Autoencoder model and a Matrix Factorization model for implicit feedback collaborative filtering.
- Build your own Factorization model and train it.
- Do negative sampling to speed-up training.
- Evaluate the trained model.

2.1 Training

2.1.1 Prepare Data For Training

The data for training/evaluation has to be in a `scipy.sparse.csr_matrix` format. Typically, the data can loaded as a `pandas.DataFrame`, that can be converted into a sparse matrix with `recoder.utils.dataframe_to_csr_matrix`.

```
import pickle

import pandas as pd
from scipy.sparse import save_npz

from recoder.utils import dataframe_to_csr_matrix


# train_df is a dataframe where each row is a user-item interaction
# and the value of that interaction
train_df = pd.read_csv('train.csv')

train_matrix, item_id_map, user_id_map = dataframe_to_csr_matrix(train_df,
                                                                 user_col='user',
                                                                 item_col='item',
```

(continues on next page)

(continued from previous page)

```
                                                                inter_col='score')

# train_matrix is a user by item interactions matrix

# item_id_map maps the original item ids into indexed item ids, such that
# the interactions with item 'whatever' can be retrieved with
# train_matrix[:, item_id_map['whatever']]

# user_id_map is like item_id_map but for users. The interactions of user 'whoever'
# can be retrieved with train_matrix[user_id_map['whoever'], :]


# you can save the sparse matrix so you don't have to
# get the sparse matrix everytime
save_npz('train.npz', matrix=train_matrix)

# also better saving the item_id_map and user_id_map
# you can do that with pickle
with open('item_id_map.dict', 'wb') as _item_id_map_file_pointer:
  pickle.dump(item_id_map, _item_id_map_file_pointer)

with open('user_id_map.dict', 'wb') as _user_id_map_file_pointer:
  pickle.dump(user_id_map, _user_id_map_file_pointer)
```

2.1.2 Autoencoder Model

```
from recoder.model import Recoder
from recoder.data import RecommendationDataset
from recoder.nn import DynamicAutoencoder

import scipy.sparse as sparse

# Load the training sparse matrix
train_matrix = sparse.load_npz('train.npz')

train_dataset = RecommendationDataset(train_matrix)

# Define your model
model = DynamicAutoencoder(hidden_layers=[200], activation_type='tanh',
                           noise_prob=0.5, sparse=True)

# Recoder takes a factorization model and trains it
recoder = Recoder(model=model, use_cuda=True,
                  optimizer_type='adam', loss='logistic')

recoder.train(train_dataset=train_dataset, batch_size=500,
              lr=1e-3, weight_decay=2e-5, num_epochs=100,
              num_data_workers=4, negative_sampling=True)
```

2.1.3 Matrix Factorization Model

Same as training Autoencoder model, just replace the Autoencoder with a Matrix Factorization Model.

```
from recoder.model import Recoder
from recoder.data import RecommendationDataset
from recoder.nn import MatrixFactorization

import scipy.sparse as sparse

# Load the training sparse matrix
train_matrix = sparse.load_npz('train.npz')

train_dataset = RecommendationDataset(train_matrix)

# Define your model
model = MatrixFactorization(embedding_size=200, activation_type='tanh',
                            dropout_prob=0.5, sparse=True)

# Recoder takes a factorization model and trains it
recoder = Recoder(model=model, use_cuda=True,
                  optimizer_type='adam', loss='logistic')

recoder.train(train_dataset=train_dataset, batch_size=500,
              lr=1e-3, weight_decay=2e-5, num_epochs=100,
              num_data_workers=4, negative_sampling=True)
```

2.1.4 Your Own Factorization Model

If you want to build your own Factorization model with the objective of reconstructing the interactions matrix, all you have to do is implement `recoder.nn.FactorizationModel` interface.

```
from recoder.model import Recoder
from recoder.data import RecommendationDataset
from recoder.nn import FactorizationModel

import scipy.sparse as sparse


# Implement your model
class YourModel(FactorizationModel):

  def init_model(self, num_items=None, num_users=None):
    # Initializes your model with the number of items and users.
    pass

  def model_params(self):
    # Returns your model parameters in a dict.
    # Used by Recoder when saving the model.
    pass

  def load_model_params(self, model_params):
    # Loads the model parameters into the model.
    # Used by Recoder when loading the model from a snapshot.
    pass

  def forward(self, input, input_users=None,
              input_items=None, target_users=None,
              target_items=None):
    # A forward pass on the model
```

(continues on next page)

(continued from previous page)

```
    # input_users are the users in the input batch
    # input_items are the items in the input batch
    # target_items are the items to be predicted
    pass


# Load the training sparse matrix
train_matrix = sparse.load_npz('train.npz')

train_dataset = RecommendationDataset(train_matrix)

# Define your model
model = YourModel()

# Recoder takes a factorization model and trains it
recoder = Recoder(model=model, use_cuda=True,
                  optimizer_type='adam', loss='logistic')

recoder.train(train_dataset=train_dataset, batch_size=500,
              lr=1e-3, weight_decay=2e-5, num_epochs=100,
              num_data_workers=4, negative_sampling=True)
```

2.1.5 Save your model

```
# You can save your model while training at different epoch checkpoints using
# model_checkpoint_prefix and checkpoint_freq params

# model state file prefix that will be appended by epoch number
model_checkpoint_prefix = 'models/model_'

recoder.train(train_dataset=train_dataset, batch_size=500,
              lr=1e-3, weight_decay=2e-5, num_epochs=100,
              num_data_workers=4, negative_sampling=True,
              model_checkpoint_prefix=model_checkpoint_prefix,
              checkpoint_freq=10)

# or you can directly call recoder.save_state
recoder.save_state(model_checkpoint_prefix)
```

2.1.6 Continue training

```
from recoder.model import Recoder
from recoder.data import RecommendationDataset
from recoder.nn import DynamicAutoencoder

import scipy.sparse as sparse

# Load the training sparse matrix
train_matrix = sparse.load_npz('train.npz')

train_dataset = RecommendationDataset(train_matrix)
```

(continues on next page)

(continued from previous page)

```
# your saved model
model_file = 'models/your_model'

# Initialize your model
# No need to set model parameters since they will be loaded
# when initializing Recoder from a saved model
model = DynamicAutoencoder()


# Initialize Recoder
recoder = Recoder(model=model, use_cuda=True)
recoder.init_from_model_file(model_file)

recoder.train(train_dataset=train_dataset, batch_size=500,
              lr=1e-3, weight_decay=2e-5, num_epochs=100,
              num_data_workers=4, negative_sampling=True)
```

2.1.7 Tips

Recoder supports training with sparse gradients. Sparse gradients training is only supported currently by the `torch.optim.SparseAdam` optimizer. This is specially helpful for training big embedding layers such as the users and items embedding layers in the Autoencoder and MatrixFactorization models. Set the `sparse` parameter in `Autoencoder` and `MatrixFactorization` to `True` in order to return sparse gradients and this can lead to 1.5-2x training speed-up. If you want to build your own model and have the embedding layers return sparse gradients, `Recoder` should be able to detect that.

2.2 Mini-batch based negative sampling

Mini-batch based negative sampling is based on the simple idea of sampling, for each user, only the negative items that the other users in the mini-batch have interacted with. This sampling procedure is biased toward popular items and in order to tune the sampling probability of each negative item, one has to tune the training batch-size. Mini-batch based negative sampling can speed-up training by 2-4x while having a small drop in recommendation performance.

- To use mini-batch based negative sampling, you have to set `negative_sampling` to `True` in `Recoder.train` and tune it with the `batch_size`:

```
recoder.train(train_dataset=train_dataset, batch_size=500,
              lr=1e-3, weight_decay=2e-5, num_epochs=100,
              num_data_workers=4, negative_sampling=True)
```

- For large datasets with large number of items, we need a large number of negative samples, hence a large batch size, which makes the batch not fit into memory and expensive to train on. In that case, we can simply generate the sparse batch with a large batch size and then slice it into smaller batches, and train on the small batches. To do this you can fix the `batch_size` to a specific value, and instead tune the `num_sampling_users` in order to increase the number of negative samples.

```
recoder.train(train_dataset=train_dataset, batch_size=500,
              negative_sampling=True, num_sampling_users=2000, lr=1e-3,
              weight_decay=2e-5, num_epochs=100, num_data_workers=4)
```

2.3 Evaluation

You can evaluate your model with different metrics. Currently, there are 3 metrics implemented: Recall, NDCG, and Average Precision. You can also implement your own `recoder.metrics.Metric`.

2.3.1 Evaluating your model while training

```
from recoder.model import Recoder
from recoder.data import RecommendationDataset
from recoder.nn import DynamicAutoencoder
from recoder.metrics import AveragePrecision, Recall, NDCG

import scipy.sparse as sparse


# Load the training sparse matrix
train_matrix = sparse.load_npz('train.npz')

# validation set. Split your val set into two splits.
# One split will be used as input to the model to
# generate predictions, and the other is which the
# model predictions will be evaluated on
val_input_matrix = sparse.load_npz('test_input.npz')
val_target_matrix = sparse.load_npz('test_target.npz')

train_dataset = RecommendationDataset(train_matrix)

val_dataset = RecommendationDataset(val_input_matrix, val_target_matrix)


# Define your model
model = DynamicAutoencoder(hidden_layers=[200], activation_type='tanh',
                           noise_prob=0.5, sparse=True)

# Initialize your metrics
metrics = [Recall(k=20, normalize=True), Recall(k=50, normalize=True),
           NDCG(k=100)]

# Recoder takes a factorization model and trains it
recoder = Recoder(model=model, use_cuda=True,
                  optimizer_type='adam', loss='logistic')

recoder.train(train_dataset=train_dataset,
              val_dataset=val_dataset, batch_size=500,
              lr=1e-3, weight_decay=2e-5, num_epochs=100,
              num_data_workers=4, negative_sampling=True,
              metrics=metrics, eval_num_recommendations=100,
              eval_freq=5)
```

2.3.2 Evaluating your model after training

```
from recoder.model import Recoder
from recoder.data import RecommendationDataset
```

(continues on next page)

(continued from previous page)

```
from recoder.nn import DynamicAutoencoder
from recoder.metrics import AveragePrecision, Recall, NDCG

import scipy.sparse as sparse


# validation set. Split your val set into two splits.
# One split will be used as input to the model to
# generate predictions, and the other is which the
# model predictions will be evaluated on
test_input_matrix = sparse.load_npz('test_input.npz')
test_target_matrix = sparse.load_npz('test_target.npz')

test_dataset = RecommendationDataset(test_input_matrix, test_target_matrix)

# your saved model
model_file = 'models/your_model'

# Initialize your model
# No need to set model parameters since they will be loaded
# when initializing Recoder from a saved model
model = DynamicAutoencoder()

# Initialize your metrics
metrics = [Recall(k=20, normalize=True), Recall(k=50, normalize=True),
           NDCG(k=100)]

# Initialize Recoder
recoder = Recoder(model=model, use_cuda=True)
recoder.init_from_model_file(model_file)

# Evaluate on the top 100 recommendations
num_recommendations = 100

recoder.evaluate(eval_dataset=test_dataset, num_recommendations=num_recommendations,
                 metrics=metrics, batch_size=500)
```

CHAPTER 3

Recoder

class recoder.model.**Recoder**(*model: recoder.nn.FactorizationModel*, *num_items=None*, *num_users=None*, *optimizer_type='sgd'*, *loss='mse'*, *loss_params=None*, *use_cuda=False*, *user_based=True*, *item_based=True*)

Module to train/evaluate a recommendation *recoder.nn.FactorizationModel*.

Parameters

- **model** (FactorizationModel) – the factorization model to train.
- **num_items** (*int, optional*) – the number of items to represent. If None, it will be computed from the first training dataset passed to `train()`.
- **num_users** (*int, optional*) – the number of users to represent. If not provided, it will be computed from the first training dataset passed to `train()`.
- **optimizer_type** (*str, optional*) – optimizer type (one of 'sgd', 'adam', 'adagrad', 'rmsprop').
- **loss** (*str or torch.nn.Module, optional*) – loss function used to train the model. If loss is a `str`, it should be *mse* for `recoder.losses.MSELoss`, *logistic* for `torch.nn.BCEWithLogitsLoss`, or *logloss* for `recoder.losses.MultinomialNLLLoss`. If `loss` is a `torch.nn.Module`, then that Module will be used as a loss function and make sure that the loss reduction is a sum reduction and not an elementwise mean.
- **loss_params** (*dict, optional*) – loss function extra params based on loss module if `loss` is a `str`. Ignored if `loss` is a `torch.nn.Module`.
- **use_cuda** (*bool, optional*) – use GPU when training/evaluating the model.
- **user_based** (*bool, optional*) – If your model is based on users or not. If True, an exception will will be raised when there are inconsistencies between the users represented in the model and the users in the training datasets.
- **item_based** (*bool, optional*) – If your model is based on items or not. If True, an exception will will be raised when there are inconsistencies between the items represented

in the model and the items in the training datasets.

init_from_model_file(*model_file*)
Initializes the model from a pre-trained model

Parameters **model_file** (*str*) – the pre-trained model file path

save_state(*model_checkpoint_prefix*)
Saves the model state in the path starting with `model_checkpoint_prefix` and appending it with the model current training epoch

Parameters **model_checkpoint_prefix** (*str*) – the model save path prefix

Returns the model state file path

train(*train_dataset*, *val_dataset=None*, *lr=0.001*, *weight_decay=0*, *num_epochs=1*, *iters_per_epoch=None*, *batch_size=64*, *lr_milestones=None*, *negative_sampling=False*, *num_sampling_users=0*, *num_data_workers=0*, *model_checkpoint_prefix=None*, *checkpoint_freq=0*, *eval_freq=0*, *eval_num_recommendations=None*, *eval_num_users=None*, *metrics=None*, *eval_batch_size=None*)
Trains the model

Parameters

- **train_dataset** (RecommendationDataset) – train dataset.
- **val_dataset** (RecommendationDataset, *optional*) – validation dataset.
- **lr** (*float, optional*) – learning rate.
- **weight_decay** (*float, optional*) – weight decay (L2 normalization).
- **num_epochs** (*int, optional*) – number of epochs to train the model.
- **iters_per_epoch** (*int, optional*) – number of training iterations per training epoch. If None, one epoch is full number of training samples in the dataset
- **batch_size** (*int, optional*) – batch size
- **lr_milestones** (*list, optional*) – optimizer learning rate epochs milestones (0.1 decay).
- **negative_sampling** (*bool, optional*) – whether to apply mini-batch based negative sampling or not.
- **num_sampling_users** (*int, optional*) – number of users to consider for sampling items. This is useful for increasing the number of negative samples in mini-batch based negative sampling while keeping the batch-size small. If 0, then num_sampling_users will be equal to batch_size.
- **num_data_workers** (*int, optional*) – number of data workers to use for building the mini-batches.
- **checkpoint_freq** (*int, optional*) – epochs frequency of saving a checkpoint of the model
- **model_checkpoint_prefix** (*str, optional*) – model checkpoint save path prefix
- **eval_freq** (*int, optional*) – epochs frequency of doing an evaluation
- **eval_num_recommendations** (*int, optional*) – num of recommendations to generate on evaluation
- **eval_num_users** (*int, optional*) – number of users from the validation dataset to use for evaluation. If None, all users in the validation dataset are used for evaluation.

- **metrics** (*list[Metric], optional*) – list of `Metric` used to evaluate the model
- **eval_batch_size** (*int, optional*) – the size of the evaluation batch

predict (*users_interactions*, *return_input=False*)
Predicts the user interactions with all items

Parameters

- **users_interactions** (`UsersInteractions`) – A batch of users' history consisting of list of `Interaction`
- **return_input** (*bool, optional*) – whether to return the dense input batch

Returns if `return_input` is `True` a tuple of the predictions and the input batch is returned, otherwise only the predictions are returned

recommend (*users_interactions*, *num_recommendations*)
Generate list of recommendations for each user in `users_hist`.

Parameters

- **users_interactions** (`UsersInteractions`) – list of users interactions.
- **num_recommendations** (*int*) – number of recommendations to generate.

Returns list of recommended items for each user in users_interactions.

Return type list

evaluate (*eval_dataset*, *num_recommendations*, *metrics*, *batch_size=1*, *num_users=None*)
Evaluates the current model given an evaluation dataset.

Parameters

- **eval_dataset** (`RecommendationDataset`) – evaluation dataset
- **num_recommendations** (*int*) – number of top recommendations to consider.
- **metrics** (*list*) – list of `Metric` to use for evaluation.
- **batch_size** (*int, optional*) – batch size of computations.

CHAPTER 4

Data

4.1 UsersInteractions

class recoder.data.**UsersInteractions**(*users*, *interactions_matrix*)

Holds the interactions of a set of users in an interactions sparse matrix

Parameters

- **users** (*np.array*) – users being represented.
- **interactions_matrix** (*scipy.sparse.csr_matrix*) – user-item interactions matrix, where `interactions_matrix[i]` correspond to the interactions of `users[i]`.

4.2 RecommendationDataset

class recoder.data.**RecommendationDataset**(*interactions_matrix*, *target_interactions_matrix=None*)

Represents a `torch.utils.data.Dataset` that iterates through the users interactions with items.

Indexing this dataset returns a *UsersInteractions* containing the interactions of the users in the index.

Parameters

- **interactions_matrix** (*scipy.sparse.csr_matrix*) – the user-item interactions matrix.
- **target_interactions_matrix** (*scipy.sparse.csr_matrix, optional*) – the target user-item interactions matrix. Mainly used for evaluation, representing the items to recommend.

4.3 RecommendationDataLoader

class recoder.data.**RecommendationDataLoader**(*dataset*, *batch_size*, *negative_sampling=False*, *num_sampling_users=0*, *num_workers=0*, *collate_fn=None*)

A DataLoader similar to torch.utils.data.DataLoader that handles *RecommendationDataset* and generate batches with negative sampling.

By default, if no collate_fn is provided, the *BatchCollator.collate()* will be used, and iterating through this dataloader will return a *Batch* at each iteration.

Parameters

- **dataset** (RecommendationDataset) – dataset from which to load the data
- **batch_size** (*int*) – number of samples per batch
- **negative_sampling** (*bool, optional*) – whether to apply mini-batch based negative sampling or not.
- **num_sampling_users** (*int, optional*) – number of users to consider for mini-batch based negative sampling. This is useful for increasing the number of negative samples while keeping the batch-size small. If 0, then num_sampling_users will be equal to batch_size.
- **num_workers** (*int, optional*) – how many subprocesses to use for data loading.
- **collate_fn** (*callable, optional*) – A function that transforms a *UsersInteractions* into a mini-batch.

4.4 Batch

class recoder.data.**Batch**(*users*, *items*, *indices*, *values*, *size*)

Represents a sparse batch of users and items interactions.

Parameters

- **users** (*torch.LongTensor*) – users that are in the batch
- **items** (*torch.LongTensor*) – items that are in the batch
- **indices** (*torch.LongTensor*) – the indices of the interactions in the sparse matrix
- **values** (*torch.LongTensor*) – the values of the interactions
- **size** (*torch.Size*) – the size of the sparse interactions matrix

4.5 BatchCollator

class recoder.data.**BatchCollator**(*batch_size*, *negative_sampling=False*)

Collator of *UsersInteractions*. It collates the users interactions into multiple *Batch* based on batch_size.

Parameters

- **batch_size** (*int*) – number of samples per batch

- **negative_sampling** (*bool, optional*) – whether to apply mini-batch based negative sampling or not.

collate(*users_interactions*)

Collates *UsersInteractions* into batches of size `batch_size`.

Parameters **users_interactions** (*UsersInteractions*) – a *UsersInteractions*.

Returns list of batches.

Return type list[*Batch*]

CHAPTER 5

Factorization Models

5.1 FactorizationModel

class recoder.nn.**FactorizationModel**

Base class for factorization models. All subclasses should implement the following methods.

init_model(*num_items=None*, *num_users=None*)

Initializes the model with the number of users and items to be represented.

Parameters

- **num_users** (*int*) – number of users to be represented in the model
- **num_items** (*int*) – number of items to be represented in the model

model_params()

Returns the model parameters. Mainly used when storing the model hyper-parameters (i.e hidden layers, activation..etc) in a snapshot file by *recoder.model.Recoder*.

Returns Model parameters.

Return type dict

load_model_params(*model_params*)

Loads the model_params into the model. Mainly used when loading the model hyper-parameters (i.e hidden layers, activation..etc) from a snapshot file of the model stored by *recoder.model.Recoder*.

Parameters **model_params** (*dict*) – model parameters

forward(*input*, *input_users=None*, *input_items=None*, *target_users=None*, *target_items=None*)

Applies a forward pass of the input on the latent factor model.

Parameters

- **input** (*torch.FloatTensor*) – the input dense matrix of user item interactions.
- **input_users** (*torch.LongTensor*) – the users represented in the input batch, where each user corresponds to a row in input based on their index.

- **input_items** (*torch.LongTensor*) – the items represented in the input batch, where each items corresponds to a column in `input` based on their index.
- **target_users** (*torch.LongTensor*) – the target users to predict. Typically, this is not used, but kept for consistency.
- **target_items** (*torch.LongTensor*) – the target items to predict.

5.2 DynamicAutoencoder

class recoder.nn.**DynamicAutoencoder**(*hidden_layers=None*, *activation_type='tanh'*, *is_constrained=False*, *dropout_prob=0.0*, *noise_prob=0.0*, *sparse=False*)

An Autoencoder module that processes variable size vectors. This is particularly efficient for cases where we only want to reconstruct sub-samples of a large sparse vector and not the whole vector, i.e negative sampling.

Let *F* be a *DynamicAutoencoder* function that reconstructs vectors of size *d*, let *X* be a matrix of size *Bxd* where *B* is the batch size, and let *Z* be a sub-matrix of *X* and *I* be a vector of any length, such that *1 <= I[i] <= d* and *Z = X[:, I]*. The reconstruction of *Z* is *F(Z, I)*. See *Examples*.

Parameters

- **hidden_layers** (*list*) – autoencoder hidden layers sizes. only the encoder layers.
- **activation_type** (*str, optional*) – activation function to use for hidden layers. all activations in torch.nn.functional are supported
- **is_constrained** (*bool, optional*) – constraining model by using the encoder weights in the decoder (tying the weights).
- **dropout_prob** (*float, optional*) – dropout probability at the bottleneck layer
- **noise_prob** (*float, optional*) – dropout (noise) probability at the input layer
- **sparse** (*bool, optional*) – if True, gradients w.r.t. to the embedding layers weight matrices will be sparse tensors. Currently, sparse gradients are only fully supported by `torch.optim.SparseAdam`.

Examples:

```
>>>> autoencoder = DynamicAutoencoder([500,100])
>>>> batch_size = 32
>>>> input = torch.rand(batch_size, 5)
>>>> input_items = torch.LongTensor([10, 126, 452, 29, 34])
>>>> output = autoencoder(input, input_items=input_items, target_items=input_
↪items)
>>>> output
   0.0850  0.9490  ...   0.2430  0.5323
   0.3519  0.4816  ...   0.9483  0.2497
        ...                  ...
   0.8744  0.8194  ...   0.5755  0.2090
   0.5006  0.9532  ...   0.8333  0.4330
  [torch.FloatTensor of size 32x5]
>>>>
>>>> # predicting a different target of items
>>>> target_items = torch.LongTensor([31, 14, 95, 49, 10, 36, 239])
>>>> output = autoencoder(input, input_items=input_items, target_items=target_
↪items)
>>>> output
```

(continues on next page)

(continued from previous page)

```
   0.5446  0.5468  ...   0.9854  0.6465
   0.0564  0.1238  ...   0.5645  0.6576
        ...                  ...
   0.0498  0.6978  ...   0.8462  0.2135
   0.6540  0.5686  ...   0.6540  0.4330
  [torch.FloatTensor of size 32x7]
>>>>
>>>> # reconstructing the whole vector
>>>> input = torch.rand(batch_size, 500)
>>>> output = autoencoder(input)
>>>> output
   0.0865  0.9054  ...   0.8987  0.0456
   0.9852  0.6540  ...   0.1205  0.8488
        ...                  ...
   0.4650  0.3540  ...   0.5646  0.5605
   0.6940  0.2140  ...   0.9820  0.5405
  [torch.FloatTensor of size 32x500]
```

init_model(*num_items=None*, *num_users=None*)

Initializes the model with the number of users and items to be represented.

Parameters

- **num_users** (*int*) – number of users to be represented in the model
- **num_items** (*int*) – number of items to be represented in the model

model_params()

Returns the model parameters. Mainly used when storing the model hyper-parameters (i.e hidden layers, activation..etc) in a snapshot file by *recoder.model.Recoder*.

Returns Model parameters.

Return type dict

load_model_params(*model_params*)

Loads the `model_params` into the model. Mainly used when loading the model hyper-parameters (i.e hidden layers, activation..etc) from a snapshot file of the model stored by *recoder.model.Recoder*.

Parameters **model_params** (*dict*) – model parameters

forward(*input*, *input_users=None*, *input_items=None*, *target_users=None*, *target_items=None*)

Applies a forward pass of the input on the latent factor model.

Parameters

- **input** (*torch.FloatTensor*) – the input dense matrix of user item interactions.
- **input_users** (*torch.LongTensor*) – the users represented in the input batch, where each user corresponds to a row in `input` based on their index.
- **input_items** (*torch.LongTensor*) – the items represented in the input batch, where each items corresponds to a column in `input` based on their index.
- **target_users** (*torch.LongTensor*) – the target users to predict. Typically, this is not used, but kept for consistency.
- **target_items** (*torch.LongTensor*) – the target items to predict.

5.3 MatrixFactorization

class recoder.nn.**MatrixFactorization**(*embedding_size*, *activation_type='none'*, *dropout_prob=0*, *sparse=False*)

Defines a Matrix Factorization model for collaborative filtering. This is particularly efficient for cases where we only want to reconstruct sub-samples of a large sparse vector and not the whole vector, i.e negative sampling.

Parameters

- **embedding_size** (*int*) – embedding size (rank) of the latent factors of users and items
- **activation_type** (*str, optional*) – activation function to be applied on the user embedding. all activations in torch.nn.functional are supported.
- **dropout_prob** (*float, optional*) – dropout probability to be applied on the user embedding
- **sparse** (*bool, optional*) – if True, gradients w.r.t. to the embedding layers weight matrices will be sparse tensors. Currently, sparse gradients are only fully supported by `torch.optim.SparseAdam`.

init_model(*num_items=None*, *num_users=None*)

Initializes the model with the number of users and items to be represented.

Parameters

- **num_users** (*int*) – number of users to be represented in the model
- **num_items** (*int*) – number of items to be represented in the model

model_params()

Returns the model parameters. Mainly used when storing the model hyper-parameters (i.e hidden layers, activation..etc) in a snapshot file by *recoder.model.Recoder*.

Returns Model parameters.

Return type dict

load_model_params(*model_params*)

Loads the `model_params` into the model. Mainly used when loading the model hyper-parameters (i.e hidden layers, activation..etc) from a snapshot file of the model stored by *recoder.model.Recoder*.

Parameters **model_params** (*dict*) – model parameters

forward(*input*, *input_users=None*, *input_items=None*, *target_users=None*, *target_items=None*)

Applies a forward pass of the input on the latent factor model.

Parameters

- **input** (*torch.FloatTensor*) – the input dense matrix of user item interactions.
- **input_users** (*torch.LongTensor*) – the users represented in the input batch, where each user corresponds to a row in `input` based on their index.
- **input_items** (*torch.LongTensor*) – the items represented in the input batch, where each items corresponds to a column in `input` based on their index.
- **target_users** (*torch.LongTensor*) – the target users to predict. Typically, this is not used, but kept for consistency.
- **target_items** (*torch.LongTensor*) – the target items to predict.

CHAPTER 6

Loss Functions

6.1 MSELoss

class recoder.losses.**MSELoss**(*confidence=0*, *reduction='elementwise_mean'*)

Computes the weighted mean squared error loss.

The weight for an observation x:

$$w = 1 + confidence \times x$$

and the loss is:

$$\ell(x, y) = w \cdot (y - x)^2$$

Parameters

- **confidence** (*float, optional*) – the weighting of positive observations.
- **reduction** (*string, optional*) – Specifies the reduction to apply to the output: 'none' | 'elementwise_mean' | 'sum'. 'none': no reduction will be applied, 'elementwise_mean': the sum of the output will be divided by the number of elements in the output, 'sum': the output will be summed. Default: 'elementwise_mean'

6.2 MultinomialNLLLoss

class recoder.losses.**MultinomialNLLLoss**(*reduction='elementwise_mean'*)

Computes the negative log-likelihood of the multinomial distribution.

$$\ell(x, y) = L = -y \cdot \log(softmax(x))$$

Parameters **reduction** (*string, optional*) – Specifies the reduction to apply to the output: 'none' | 'elementwise_mean' | 'sum'. 'none': no reduction will be applied, 'elementwise_mean': the sum of the output will be divided by the number of elements in the output, 'sum': the output will be summed. Default: 'elementwise_mean'

CHAPTER 7

Embeddings Indices

7.1 EmbeddingsIndex

class recoder.embedding.**EmbeddingsIndex**

An abstract Embeddings Index from which to fetch embeddings and execute nearest neighbor search on the items represented by the embeddings

All `EmbeddingsIndex` should implement this interface.

get_embedding(*embedding_id*)

Returns the embedding of the item `embedding_id`

get_nns_by_id(*embedding_id*, *n*)

Returns the `n` nearest neighbors of the `embedding_id`

get_nns_by_embedding(*embedding*, *n*)

Returns the `n` nearest neighbors of the `embedding`

get_similarity(*id1*, *id2*)

Returns the similarity between item `id1` and item `id2`

7.2 AnnoyEmbeddingsIndex

class recoder.embedding.**AnnoyEmbeddingsIndex**(*embeddings=None*, *id_map=None*, *n_trees=10*, *search_k=-1*, *include_distances=False*)

An `EmbeddingsIndex` based on `AnnoyIndex` [1] to efficiently execute nearest neighbors search with trade off in accuracy.

The similarity between items is the cosine similarity.

Parameters

- **embeddings** (*numpy.array, optional*) – the matrix that holds the embeddings of shape (number of items * embedding size). Required to build the index.

- **id_map** (*dict, optional*) – A dict that maps the items original ids to the indices of the embeddings. Useful to fetch and do nearest neighbor search on the original items ids. If not provided, it will simply be an identity map.
- **n_trees** (*int, optional*) – n_trees parameter used to build AnnoyIndex.
- **search_k** (*int, optional*) – search_k parameter used to search the AnnoyIndex for nearest items.
- **include_distances** (*bool, optional*) – include distances in the result returned on nearest search

[1]: https://github.com/spotify/annoy

build(*index_file=None*)
Builds the embeddings index, and stores it in `index_file` if provided.

Parameters **index_file** (*str, optional*) – the index file path where to save the index. Note: The annoy index file is stored in a separate file, which should be in the same directory as `index_file`.

load(*index_file*)
Loads the embeddings index from a saved index file.

Parameters **index_file** (*str*) – the index file path to load the state of the index. Note: The annoy index file is stored in a separate file, which should be in the same directory as index_file.

get_embedding(*embedding_id*)
Returns the embedding of the item `embedding_id`

get_nns_by_id(*embedding_id*, *n*)
Returns the `n` nearest neighbors of the `embedding_id`

get_nns_by_embedding(*embedding*, *n*)
Returns the `n` nearest neighbors of the `embedding`

get_similarity(*id1*, *id2*)
Returns the similarity between item `id1` and item `id2`

7.3 MemCacheEmbeddingsIndex

class recoder.embedding.**MemCacheEmbeddingsIndex**(*embedding_index*)
Caches `EmbeddingsIndex` nearest neighbor search results for each item in memory to reduce computations.

Parameters **embedding_index** (`EmbeddingsIndex`) – the EmbeddingsIndex to hit on cache misses.

get_embedding(*embedding_id*)
Returns the embedding of the item `embedding_id`

get_nns_by_embedding(*embedding*, *n*)
Returns the `n` nearest neighbors of the `embedding`

get_nns_by_id(*embedding_id*, *n*)
Returns the `n` nearest neighbors of the `embedding_id`

get_similarity(*id1*, *id2*)
Returns the similarity between item `id1` and item `id2`

CHAPTER 8

Metrics

8.1 Metric

class recoder.metrics.**Metric**(*metric_name*)

A Base class for metrics. All metrics should implement the `evaluate` function.

Parameters **metric_name** (*str*) – metric name. useful for representing it as string (printing) and hashing.

evaluate(*x*, *y*)

Evaluates the recommendations with respect to the items the user interacted with.

Parameters

- **x** (*list*) – items recommended for the user
- **y** (*list*) – items the user interacted with

Returns metric value

Return type float

8.2 Recall

class recoder.metrics.**Recall**(*k*, *normalize=True*)

Computes the recall @ K of the recommended items.

Parameters

- **k** (*int*) – the cut position of the recommended list
- **normalize** (*bool, optional*) – if True, normalize the value to 1 (divide by k) if k is less than the number of items in the user interactions, otherwise normalize only by number of items in the user interactions.

evaluate(*x*, *y*)
Evaluates the recommendations with respect to the items the user interacted with.

Parameters

- **x** (*list*) – items recommended for the user
- **y** (*list*) – items the user interacted with

Returns metric value

Return type float

8.3 NDCG

class recoder.metrics.**NDCG**(*k*)
Computes the normalized discounted cumulative gain @ K of the recommended items.

Parameters **k** (*int*) – the cut position of the recommended list

evaluate(*x*, *y*)
Evaluates the recommendations with respect to the items the user interacted with.

Parameters

- **x** (*list*) – items recommended for the user
- **y** (*list*) – items the user interacted with

Returns metric value

Return type float

8.4 AveragePrecision

class recoder.metrics.**AveragePrecision**(*k*, *normalize=True*)
Computes the average precision @ K of the recommended items.

Parameters

- **k** (*int*) – the cut position of the recommended list
- **normalize** (*bool, optional*) – if True, normalize the value to 1 (divide by k) if k is less than the number of items the user interacted with, otherwise normalize only by number of items the user interacted with.

evaluate(*x*, *y*)
Evaluates the recommendations with respect to the items the user interacted with.

Parameters

- **x** (*list*) – items recommended for the user
- **y** (*list*) – items the user interacted with

Returns metric value

Return type float

8.5 RecommenderEvaluator

class recoder.metrics.**RecommenderEvaluator**(*recommender*, *metrics*)

Evaluates a *recoder.recommender.Recommender* given a set of *Metric*

Parameters

- **recommender** (Recommender) – the Recommender to evaluate
- **metrics** (*list*) – list of metrics used to evaluate the recommender

evaluate(*eval_dataset*, *batch_size=1*, *num_users=None*, *num_workers=0*)

Evaluates the recommender with an evaluation dataset.

Parameters

- **eval_dataset** (RecommendationDataset) – the dataset to use in evaluating the model
- **batch_size** (*int*) – the size of the users batch passed to the recommender
- **num_users** (*int, optional*) – the number of users from the dataset to evaluate on. If None, evaluate on all users
- **num_workers** (*int, optional*) – the number of workers to use on evaluating the recommended items. This is useful if the recommender runs on GPU, so the evaluation can run in parallel.

Returns A dict mapping each metric to the list of the metric values on each user in the dataset.

Return type dict

CHAPTER 9

Recommenders

9.1 Recommender

class recoder.recommender.**Recommender**

Base Recommender that provide recommendations given users history of interactions. All Recommenders should implement the `recommend` function.

recommend(*users_hist*)

Recommends a list of items for each user list of `recoder.data.UserInteractions`.

Parameters **users_hist** (*list*) – list of users list of `recoder.data.UserInteractions`.

Returns items recommended for each user

Return type list

9.2 SimilarityRecommender

class recoder.recommender.**SimilarityRecommender**(*embeddings_index: recoder.embedding.EmbeddingsIndex*, *num_recommendations*, *n=1*, *scale=1*)

Recommends items based on similarity search of the items in the user list of `recoder.data.UserInteractions`.

Implementation based on [1].

Note: This still needs improvement and optimization, and its implementation might change.

Parameters

- **embeddings_index** (`EmbeddingsIndex`) – the embeddings index used to fetch embeddings and do nearest neighbor search.

- **num_recommendations** (*int*) – number of recommendations to generate for each user. Note: the number of recommendations requirement is not necessarily satisfied.
- **n** (*int, optional*) – number of similar items to retrieve for every item in user interactions.
- **scale** (*int, optional*) – how much to scale the similarity between two items

[1]: Fabio Aiolli. 2013. Efficient top-n recommendation for very large scale binary rated datasets. In Proceedings of the 7th ACM conference on Recommender systems (RecSys '13). ACM, New York, NY, USA, 273-280. DOI=http://dx.doi.org/10.1145/2507157.2507189

recommend(*users_hist*)

Recommends a list of items for each user list of `recoder.data.UserInteractions`.

Parameters **users_hist** (*list*) – list of users list of `recoder.data.UserInteractions`.

Returns items recommended for each user

Return type list

9.3 InferenceRecommender

class recoder.recommender.**InferenceRecommender**(*model*, *num_recommendations*)

Recommends items based on the predictions by a *recoder.model.Recoder* model.

Parameters

- **model** (Recoder) – model used to predict recommendations
- **num_recommendations** (*int*) – number of recommendations to generate for each user.

recommend(*users_hist*)

Recommends a list of items for each user list of `recoder.data.UserInteractions`.

Parameters **users_hist** (*list*) – list of users list of `recoder.data.UserInteractions`.

Returns items recommended for each user

Return type list

CHAPTER 10

Utils

recoder.utils.**normalize**(*x*, *axis=None*)

Returns the normalization of *x* along *axis*.

Parameters

- **x** (*np.array*) – matrix or vector
- **axis** (*int, optional*) – the axis along which to compute the normalization

recoder.utils.**dataframe_to_csr_matrix**(*dataframe*, *user_col*, *item_col*, *inter_col*, *item_id_map=None*, *user_id_map=None*)

Converts a `pandas.DataFrame` of users and items interactions into a `scipy.sparse.csr_matrix`.

This function returns a tuple of the interactions sparse matrix, a *dict* that maps from original item ids in the dataframe to the 0-based item ids, and similarly a *dict* that maps from original user ids in the dataframe to the 0-based user ids.

Parameters

- **dataframe** (*pandas.DataFrame*) – A dataframe containing users and items interactions
- **user_col** (*str*) – users column name
- **item_col** (*str*) – items column name
- **inter_col** (*str*) – user-item interaction value column name
- **item_id_map** (*dict, optional*) – A dictionary mapping from original item ids into 0-based item ids. If not given, the map will be generated using the items column in the dataframe
- **user_id_map** (*dict, optional*) – A dictionary mapping from original user ids into 0-based user ids. If not given, the map will be generated using the users column in the dataframe

Returns A tuple of the *csr_matrix*, a `dict` *item_id_map*, and a `dict` *user_id_map*

Return type tuple

CHAPTER 11

Indices and tables

- genindex
- modindex
- search

Index

I

L

M

N

P

R

S

T

U