ml-1m dl (new version)

- Here I will build the dataset with labeled gender and ages, then after splitting, save the dataset
- Do not use sex / age in this version

参考: [Collaborative Filterting on MovieLens Dataset (https://github.com/devforfu/pytorch_playground/blob/master/movielens.ipynb)](https://github.com/devforfu/pytorch_playground/blob/master/movielens.ipynb)

In [1]:
```
# import io
# import os
import math
import copy
import pickle
# import zipfile
# from textwrap import wrap
from pathlib import Path
from itertools import zip_longest
from collections import defaultdict
# from urllib.error import URLError
# from urllib.request import urlopen

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split, KFold

import torch
from torch import nn
from torch import optim
from torch.nn import functional as F
from torch.optim.lr_scheduler import _LRScheduler

from time import time
from collections import defaultdict

%matplotlib inline
```

In [2]:
```
def set_random_seed(state=1):
    gens = (np.random.seed, torch.manual_seed, torch.cuda.manual_seed)
    for set_state in gens:
        set_state(state)


RANDOM_STATE = 1
set_random_seed(RANDOM_STATE)
```

In [3]:
```
# 加载数据集
df = pd.read_csv("data/ml-1m_merged/ml-1m_merged.csv")
print(df.shape)
df.head()
```

```
(1000209, 7)
```

Out[3]:

	movie_id	movie_title	user_id	age	sex	occupation	rating
0	1	Toy Story (1995)	1	1	F	10	5
1	48	Pocahontas (1995)	1	1	F	10	5
2	150	Apollo 13 (1995)	1	1	F	10	5
3	260	Star Wars: Episode IV - A New Hope (1977)	1	1	F	10	4
4	527	Schindler's List (1993)	1	1	F	10	5

In [4]:
```
# relabel sex
sex_dict = {'F':0, 'M':1}
age_dict = {1:0, 18:1, 25:2, 35:3, 45:4, 50:5, 56:6}
```

In [5]:
```
df['sex_index'] = df['sex'].map(sex_dict)
df['age_index'] = df['age'].map(age_dict)
df.head()
```

Out[5]:

	movie_id	movie_title	user_id	age	sex	occupation	rating	sex_index	age_index
0	1	Toy Story (1995)	1	1	F	10	5	0	0
1	48	Pocahontas (1995)	1	1	F	10	5	0	0
2	150	Apollo 13 (1995)	1	1	F	10	5	0	0
3	260	Star Wars: Episode IV - A New Hope (1977)	1	1	F	10	4	0	0
4	527	Schindler's List (1993)	1	1	F	10	5	0	0

In [20]:
```
# save this dataset
df.to_csv('ml-1m_dl.csv', index=0)
```

构建data / label

In [8]:
```
# 构建data与label
def create_dataset(df):
    n_users = len(set(df['user_id']))
    n_movies = len(set(df['movie_id']))
    X = df[['user_id','movie_id','sex_index','age_index']]
    y = df['rating'].astype(np.float32)
    return (n_users, n_movies), (X, y)
```

In [9]:
```
# 这里注意movie id --> 之后我设置成4000好啦
(n_users, n_movies), (X, y) = create_dataset(df)
print("{} users, {} movies".format(n_users, n_movies))
print("user id: max {}, min {}".format(X['user_id'].max(), X['user_id'].min()))
print("movie id: max {}, min {}".format(X['movie_id'].max(), X['movie_id'].min()))
print(X.shape)
print(y.shape)
```

```
6040 users, 3706 movies
user id: max 6040, min 1
movie id: max 3952, min 1
(1000209, 4)
(1000209,)
```

batch-wise data iterator

In [10]:
```
# batch-wise data iterator
class ReviewsIterator:
    def __init__(self, X, y, batch_size=32, shuffle=True):
        X, y = np.asarray(X), np.asarray(y)

        if shuffle:
            index = np.random.permutation(X.shape[0])
            X, y = X[index], y[index]

        self.X = X
        self.y = y
        self.batch_size = batch_size
        self.shuffle = shuffle
        self.n_batches = int(math.ceil(X.shape[0] // batch_size))
        self._current = 0

    def __iter__(self):
        return self

    def __next__(self):
        return self.next()

    def next(self):
        if self._current >= self.n_batches:
            raise StopIteration()
        k = self._current
        self._current += 1
        bs = self.batch_size
        return self.X[k*bs:(k + 1)*bs], self.y[k*bs:(k + 1)*bs]


def batches(X, y, bs=32, shuffle=True):
    for xb, yb in ReviewsIterator(X, y, bs, shuffle):
        xb = torch.LongTensor(xb)
        yb = torch.FloatTensor(yb)
        yield xb, yb.view(-1, 1)
```

In [11]:
```
for x_batch, y_batch in batches(X, y, bs=4):
    print(x_batch)
    print(y_batch)
    break
```

```
tensor([[5837, 2353,    1,    2],
        [2242, 3114,    1,    1],
        [ 103, 1801,    1,    4],
        [1635, 1215,    1,    2]])
tensor([[4.],
        [5.],
        [3.],
        [5.]])
```

In [ ]:

define the network

In [12]:
```
class EmbeddingNet(nn.Module):
    def __init__(self, n_users, n_movies, n_factors=50, embedding_dropout=0.02, hidden=10, dropouts=0.2):
        super().__init__()
        hidden = get_list(hidden)
        dropouts = get_list(dropouts)
        n_last = hidden[-1]
        def gen_layers(n_in):
            """
            A generator that yields a sequence of hidden layers and
            their activations/dropouts.

            Note that the function captures `hidden` and `dropouts`
            values from the outer scope.
            """
            nonlocal hidden, dropouts
            assert len(dropouts) <= len(hidden)
            for n_out, rate in zip_longest(hidden, dropouts):
                yield nn.Linear(n_in, n_out)
                yield nn.ReLU()
                if rate is not None and rate > 0.:
                    yield nn.Dropout(rate)
                n_in = n_out


        self.u = nn.Embedding(n_users+1 , n_factors) # hard code
        self.m = nn.Embedding(4000, n_factors)  # hardcode

        self.drop = nn.Dropout(embedding_dropout)
        self.hidden = nn.Sequential(*list(gen_layers(n_factors * 2)))
        self.fc = nn.Linear(n_last, 1)
        self._init()
    def forward(self, users, movies, minmax=None):
        uu = self.u(users)
        mm = self.m(movies)
        features = torch.cat([uu, mm], dim=1)
        x = self.drop(features)
        x = self.hidden(x)
        out = torch.sigmoid(self.fc(x))
        if minmax is not None:
            min_rating, max_rating = minmax
            out = out*(max_rating - min_rating + 1) + min_rating - 0.5
        return out

    def _init(self):
        def init(m):
            if type(m) == nn.Linear:
                torch.nn.init.xavier_uniform_(m.weight)
                m.bias.data.fill_(0.01)
        self.u.weight.data.uniform_(-0.05, 0.05)
        self.m.weight.data.uniform_(-0.05, 0.05)
        self.hidden.apply(init)
        init(self.fc)

def get_list(n):
    if isinstance(n, (int, float)):
        return [n]
    elif hasattr(n, '__iter__'):
        return list(n)
    raise TypeError('layers configuraiton should be a single number or a list of numbers')
```

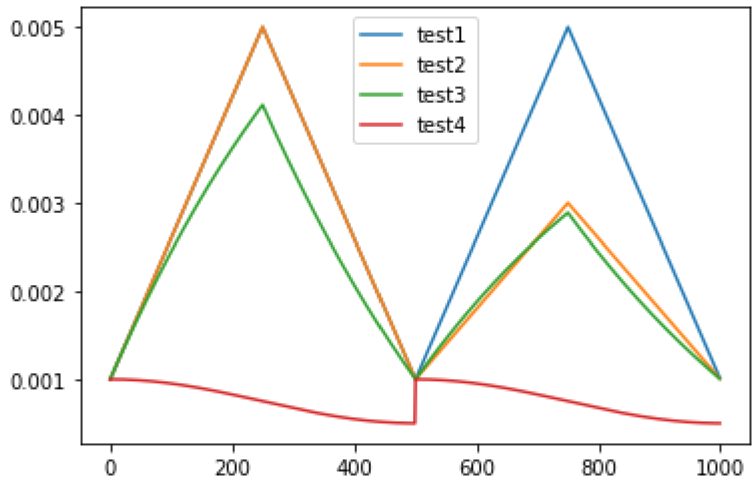
In [13]:
```
# test
testnet = EmbeddingNet(n_users, n_movies, n_factors=150, hidden=100, dropouts=0.5)
print(testnet)
testnet = EmbeddingNet(n_users, n_movies, n_factors=150, hidden=[100, 200, 300], dropouts=[0.25, 0.5])
print(testnet)
```

```
EmbeddingNet(
  (u): Embedding(6041, 150)
  (m): Embedding(4000, 150)
  (drop): Dropout(p=0.02, inplace=False)
  (hidden): Sequential(
    (0): Linear(in_features=300, out_features=100, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.5, inplace=False)
  )
  (fc): Linear(in_features=100, out_features=1, bias=True)
)
EmbeddingNet(
  (u): Embedding(6041, 150)
  (m): Embedding(4000, 150)
  (drop): Dropout(p=0.02, inplace=False)
  (hidden): Sequential(
    (0): Linear(in_features=300, out_features=100, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.25, inplace=False)
    (3): Linear(in_features=100, out_features=200, bias=True)
    (4): ReLU()
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=200, out_features=300, bias=True)
    (7): ReLU()
  )
  (fc): Linear(in_features=300, out_features=1, bias=True)
)
```

LR scheduler

In [16]:
```
class CyclicLR(_LRScheduler):
    def __init__(self, optimizer, schedule, last_epoch=-1):
        assert callable(schedule)
        self.schedule = schedule
        super().__init__(optimizer, last_epoch)
    def get_lr(self):
        return [self.schedule(self.last_epoch, lr) for lr in self.base_lrs]

def triangular(step_size, max_lr, method='triangular', gamma=0.99):

    def scheduler(epoch, base_lr):
        period = 2 * step_size
        cycle = math.floor(1 + epoch/period)
        x = abs(epoch/step_size - 2*cycle + 1)
        delta = (max_lr - base_lr)*max(0, (1 - x))

        if method == 'triangular':
            pass  # we've already done
        elif method == 'triangular2':
            delta /= float(2 ** (cycle - 1))
        elif method == 'exp_range':
            delta *= (gamma**epoch)
        else:
            raise ValueError('unexpected method: %s' % method)

        return base_lr + delta

    return scheduler

def cosine(t_max, eta_min=0):

    def scheduler(epoch, base_lr):
        t = epoch % t_max
        return eta_min + (base_lr - eta_min)*(1 + math.cos(math.pi*t/t_max))/2

    return scheduler

def plot_lr(schedule, label):
    ts = list(range(1000))
    y = [schedule(t, 0.001) for t in ts]
    plt.plot(ts, y, label=label)
```

In [17]:
```
plot_lr(triangular(250, 0.005),'test1')
plot_lr(triangular(250, 0.005, 'triangular2'),'test2')
plot_lr(triangular(250, 0.005, 'exp_range', gamma=0.999),'test3')
plot_lr(cosine(t_max=500, eta_min=0.0005),'test4')
plt.legend()
plt.show()
```



Split and save dataset

In [19]:
```
# split the dataset: 80-20
X_train, X_valid, y_train, y_valid = train_test_split(X, y, test_size=0.2, random_state=RANDOM_STATE)
datasets = {'train': (X_train, y_train), 'val': (X_valid, y_valid)}
dataset_sizes = {'train': len(X_train), 'val': len(X_valid)}
print(X_train['user_id'].max(), X_train['user_id'].min())
print(X_train['movie_id'].max(), X_train['movie_id'].min())
print(dataset_sizes)
X_train.head()
```

```
6040 1
3952 1
{'train': 800167, 'val': 200042}
```

Out[19]:

	user_id	movie_id	sex_index	age_index
529184	5530	1488	0	1
341591	3600	609	1	3
470922	4889	1291	1	1
630004	5837	1573	1	2
131938	1354	377	0	2

In [21]:
```
# save as pickle
with open("ml-1m_dl.pkl", 'wb') as ppp:
    pickle.dump(datasets, ppp)
```

In [22]:
```
# load
datasets_reload = pickle.load(open('ml-1m_dl.pkl','rb'))
datasets_reload['val'][1]
```

Out[22]:
```
630120    4.0
229398    5.0
758377    3.0
159240    5.0
254252    4.0
         ...
875199    4.0
743921    4.0
527163    4.0
623363    3.0
120098    3.0
Name: rating, Length: 200042, dtype: float32
```

Training

In [23]:
```
def train_model(datasets, model, lr, wd, bs, n_epochs, patience):
    minmax = (1.0, 5.0)
    device = torch.device('cpu')

    # Training
    no_improvements = 0
    best_loss = np.inf
    best_weights = None
    history = []
    lr_history = []

    start_time = time()
    model.to(device)
    criterion = nn.MSELoss(reduction='sum')
    optimizer = optim.Adam(model.parameters(), lr=lr, weight_decay=wd)
    iterations_per_epoch = int(math.ceil(dataset_sizes['train'] // bs))
    scheduler = CyclicLR(optimizer, cosine(t_max=iterations_per_epoch * 2, eta_min=lr/10))

    start_time = time()

    for epoch in range(n_epochs):
        stats = {'epoch': epoch + 1, 'total': n_epochs}

        for phase in ('train', 'val'):
            training = phase == 'train'
            running_loss = 0.0
            n_batches = 0

            for batch in batches(*datasets[phase], shuffle=training, bs=bs):
                x_batch, y_batch = [b.to(device) for b in batch] # [2000, 4], [2000, 1]
                optimizer.zero_grad()
                # compute gradients only during 'train' phase
                with torch.set_grad_enabled(training):
                    outputs = model(x_batch[:, 0], x_batch[:, 1], minmax)
                    loss = criterion(outputs, y_batch)
                    # don't update weights and rates when in 'val' phase
                    if training:
                        loss.backward()
                        optimizer.step()
                        scheduler.step()
                        lr_history.extend(scheduler.get_lr())
                running_loss += loss.item()

            epoch_loss = running_loss / dataset_sizes[phase]
            stats[phase] = epoch_loss

            # early stopping: save weights of the best model so far
            if phase == 'val':
                if epoch_loss < best_loss:
                    print('loss improvement on epoch: %d' % (epoch + 1))
                    best_loss = epoch_loss
                    best_weights = copy.deepcopy(model.state_dict())
                    no_improvements = 0
                else:
                    no_improvements += 1

        history.append(stats)
        cost_time = (time() - start_time) / 60.
        print('[{:03d}/{:03d}]|train {:.4f}|val {:.4f}|Time {:.2f}mins'.format(
                                                                    stats['epoch'], stats['total'],
                                                                    stats['train'], stats['val'], cost_ti
me))
        if no_improvements >= patience:
            print('early stopping after epoch {:03d}'.format(stats['epoch']))
            break
    return best_weights
```

In [35]:
```
def get_result_df(datasets, model, best_weights, bs, save_path=None):
    '''
    model_parameter1_best.weights
    '''
    minmax = (1.0, 5.0)
    device = torch.device('cpu')
    model.load_state_dict(best_weights)
    groud_truth, predictions = [], []

    val_size = len(datasets['val'][0])
    # print("Total val size: {}".format(val_size))

    with torch.no_grad():
        for batch in batches(*datasets['val'], shuffle=False, bs=bs):
            x_batch, y_batch = [b.to(device) for b in batch]
            outputs = model(x_batch[:, 0], x_batch[:, 1], minmax)
            groud_truth.extend(y_batch.tolist())
            predictions.extend(outputs.tolist())

        last_num = val_size % bs
        # print("Last num: {}".format(last_num))
        dataset_last = (datasets['val'][0][-last_num:], datasets['val'][1][-last_num:])
        # print("Last dataset: {}".format(len(dataset_last[0])))
        for batch in batches(*dataset_last, shuffle=False, bs=1):
            x_batch, y_batch = [b.to(device) for b in batch]
            outputs = model(x_batch[:, 0], x_batch[:, 1], minmax)
            groud_truth.extend(y_batch.tolist())
            predictions.extend(outputs.tolist())

    groud_truth = np.asarray(groud_truth).ravel()
    predictions = np.asarray(predictions).ravel()

    assert(predictions.shape[0] == val_size)

    final_loss = np.sqrt(np.mean((predictions - groud_truth)**2))
    print("RMSE: {:.4f}".format(final_loss))

    df_final = pd.DataFrame(datasets['val'][0])[['user_id','movie_id']]
    df_final['truth'] = datasets['val'][1]
    df_final['pred'] = predictions

    if save_path is not None:
        print("Save weight to:{}".format(save_path))
        with open(save_path, 'wb') as file:
            pickle.dump(best_weights, file)

    return df_final # note that here the sex and age is not included
```

In [26]:
```
def get_precision_recall(df_final, k=10, threshold=3.5):
    # map prediction to each user --> similar to top n
    # {id:(pred, truth)}
    user_pred_truth = defaultdict(list)
    for row in df_final.itertuples():
        _, user_id, movie_id, truth, pred = row
        user_pred_truth[user_id].append((pred, truth))

    precisions = dict()
    recalls = dict()

    for user_id, user_ratings in user_pred_truth.items():
        # Sort user ratings by estimated value
        user_ratings.sort(key=lambda x: x[0], reverse=True)

        # Number of relevant items
        n_rel = sum((true_r >= threshold) for (_, true_r) in user_ratings)

        # Number of recommended items in top k
        n_rec_k = sum((est >= threshold) for (est, _) in user_ratings[:k])

        # Number of relevant and recommended items in top k
        n_rel_and_rec_k = sum(((true_r >= threshold) and (est >= threshold))
                              for (est, true_r) in user_ratings[:k])

        # Precision@K: Proportion of recommended items that are relevant
        precisions[user_id] = n_rel_and_rec_k / n_rec_k if n_rec_k != 0 else 1

        # Recall@K: Proportion of relevant items that are recommended
        recalls[user_id] = n_rel_and_rec_k / n_rel if n_rel != 0 else 1

    # mean precision and recall
    mean_precision = sum(prec for prec in precisions.values()) / len(precisions)
    mean_recall = sum(rec for rec in recalls.values()) / len(recalls)
    print("Prec10 {:.4f}|Rec10 {:.4f}".format(mean_precision, mean_recall))
```

In [27]:
```
# get topn
def get_top_n(df_final, n=10):
    """
    key: user_id
    value: his top 10 highest movies as well as ratings
    """
    # map predictions to each user
    top_n = defaultdict(list)
    for row in df_final.itertuples():
        _, user_id, movie_id, truth, pred = row
        top_n[user_id].append((movie_id, pred))

    # sort the pred for each user
    for user_id, pred_ratings in top_n.items():
        pred_ratings.sort(key=lambda x: x[1], reverse=True)
        top_n[user_id] = pred_ratings[:n]
    return top_n

# i = 0
# top_n = get_top_n(df_final, n=10)
# for user_id, pred_ratings in top_n.items():
#     print("User id: {}".format(user_id))
#     for (movie_id, rating) in pred_ratings:
#         print("Movie {:<5d}|Rating {:.2f}".format(movie_id, rating))
#     print("----- -----")
#     i += 1
#     if i > 0:
#         break
```

In [28]:
```
def get_train_pred_top10(df, df_final, datasets, user_id=1635, n=10):
    """
    df: the original df --> contain movie name
    df_final: the final df with predicted ratings
    datasets: the datasets with training and testing datasets
    user_id: the user id to be queried
    n: top_n
    """
    # step 1, get top n from df_final
    top_n = get_top_n(df_final, n)
    assert(user_id in top_n), "user_id {} is not in testing data, try another user such as 1635".format(user_
id)
    pred_ratings = top_n[user_id]

    # step 2: user information
    user = df[df['user_id'] == user_id]
    age = list(set(user['age']))[0]
    sex = list(set(user['sex']))[0]
    info = "User {}, age {}, {}, ".format(user_id, age, sex)

    # step 2, build df_train
    df_train = pd.DataFrame(datasets['train'][0])
    df_train['rating'] = datasets['train'][1]

    # step 3, find all movies user_id has been rated 5
    # df_refined = df_train[(df_train['user_id'] == user_id) & (df_train['rating'] == 5)]
    df_refined = df_train[df_train['user_id'] == user_id]
    movie_id_sets_train = set(df_refined['movie_id'])
    info = "{} has rated {} movies in training set\n".format(info, len(movie_id_sets_train))
    print(info)

    # step 4: get the top n
    print("===== ===== ===== ===== ===== =====")
    print("\nTop {} recommendations\n".format(n))
    for (movie_id, rating) in pred_ratings:
        movie_name = list(set(df[df['movie_id'] == movie_id]['movie_title']))[0]
        info = "ID {:<4d}|Rating {:2f}|{}".format(movie_id, rating, movie_name)
        if movie_id in movie_id_sets_train:
            info = "{}, but this movie has been rated during training!!!".format(info)
        print(info)
```

In [ ]:

In [ ]:

Now let's begin

parameter set 1

In [ ]:
```
# hyper parameters

# model
n_factors = 150
hidden = [500,500,500]
embedding_dropout = 0.05
dropouts = [0.5,0.5,0.25]

# training
lr = 1e-3
wd = 1e-5
bs = 2000
n_epochs = 100
patience = 10
 # torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
# build network
model = EmbeddingNet(n_users, n_movies,
                     n_factors=n_factors,
                     hidden=hidden,
                     embedding_dropout=embedding_dropout,
                     dropouts=dropouts)
best_weights = train_model(datasets, model, lr, wd, bs, n_epochs, patience)
```

In [33]:
```
df_final = get_result_df(datasets, model, best_weights, bs, save_path='noSex_noAge_para1.weights')
get_precision_recall(df_final, k=10, threshold=3.5)
```

```
Total val size: 200042
Last dataset: 42
RMSE: 0.8818
Save weight to:noSex_noAge_para1.weights
Prec10 0.7930|Rec10 0.5582
```

parameter set 2

In [36]:
```
# model --> better than para 1
n_factors = 200
hidden = [750,750,750]
embedding_dropout = 0.05
dropouts = [0.5,0.5,0.25]
# training
lr = 1e-3
wd = 1e-5
bs =2000
n_epochs = 100
patience = 10

model = EmbeddingNet(n_users, n_movies,
                     n_factors=n_factors,
                     hidden=hidden,
                     embedding_dropout=embedding_dropout,
                     dropouts=dropouts)
best_weights = train_model(datasets, model, lr, wd, bs, n_epochs, patience)
df_final = get_result_df(datasets, model, best_weights, bs, save_path='noSex_noAge_para2.weights')
get_precision_recall(df_final, k=10, threshold=3.5)
```

```
loss improvement on epoch: 1
[001/100]|train 0.8862|val 0.8171|Time 0.57mins
loss improvement on epoch: 2
[002/100]|train 0.7928|val 0.7993|Time 1.05mins
loss improvement on epoch: 3
[003/100]|train 0.7955|val 0.7892|Time 1.55mins
loss improvement on epoch: 4
[004/100]|train 0.7508|val 0.7772|Time 2.12mins
[005/100]|train 0.7624|val 0.7795|Time 2.61mins
loss improvement on epoch: 6
[006/100]|train 0.7237|val 0.7755|Time 3.11mins
[007/100]|train 0.7364|val 0.7765|Time 3.71mins
[008/100]|train 0.6838|val 0.7808|Time 4.39mins
[009/100]|train 0.6959|val 0.7772|Time 4.94mins
[010/100]|train 0.6279|val 0.7967|Time 5.58mins
[011/100]|train 0.6466|val 0.7858|Time 6.16mins
[012/100]|train 0.5781|val 0.8107|Time 6.78mins
[013/100]|train 0.6021|val 0.8012|Time 7.38mins
[014/100]|train 0.5394|val 0.8258|Time 8.00mins
[015/100]|train 0.5667|val 0.8097|Time 8.54mins
[016/100]|train 0.5097|val 0.8345|Time 9.13mins
early stopping after epoch 016
RMSE: 0.8811
Save weight to:noSex_noAge_para2.weights
Prec10 0.7917|Rec10 0.5599
```

parameter set 3

In [37]:
```
# model --> same hidden with 1, but larger n_factors
# It seems that the hidden dim matters
n_factors = 200
hidden = [500,500,500]
embedding_dropout = 0.05
dropouts = [0.5,0.5,0.25]
# training
lr = 1e-3
wd = 1e-5
bs =2000
n_epochs = 100
patience = 10

model = EmbeddingNet(n_users, n_movies,
                     n_factors=n_factors,
                     hidden=hidden,
                     embedding_dropout=embedding_dropout,
                     dropouts=dropouts)
best_weights = train_model(datasets, model, lr, wd, bs, n_epochs, patience)
df_final = get_result_df(datasets, model, best_weights, bs, save_path='noSex_noAge_para3.weights')
get_precision_recall(df_final, k=10, threshold=3.5)
```

```
loss improvement on epoch: 1
[001/100]|train 0.8899|val 0.8198|Time 0.42mins
loss improvement on epoch: 2
[002/100]|train 0.7971|val 0.8048|Time 0.80mins
loss improvement on epoch: 3
[003/100]|train 0.8009|val 0.7957|Time 1.17mins
loss improvement on epoch: 4
[004/100]|train 0.7581|val 0.7834|Time 1.59mins
loss improvement on epoch: 5
[005/100]|train 0.7662|val 0.7820|Time 2.05mins
loss improvement on epoch: 6
[006/100]|train 0.7254|val 0.7791|Time 2.43mins
loss improvement on epoch: 7
[007/100]|train 0.7365|val 0.7786|Time 2.79mins
[008/100]|train 0.6837|val 0.7840|Time 3.14mins
loss improvement on epoch: 9
[009/100]|train 0.6965|val 0.7779|Time 3.52mins
[010/100]|train 0.6363|val 0.7949|Time 3.87mins
[011/100]|train 0.6557|val 0.7858|Time 4.24mins
[012/100]|train 0.5965|val 0.8082|Time 4.58mins
[013/100]|train 0.6195|val 0.7991|Time 4.93mins
[014/100]|train 0.5650|val 0.8193|Time 5.27mins
[015/100]|train 0.5906|val 0.8089|Time 5.67mins
[016/100]|train 0.5406|val 0.8287|Time 6.05mins
[017/100]|train 0.5668|val 0.8192|Time 6.44mins
[018/100]|train 0.5216|val 0.8380|Time 6.82mins
[019/100]|train 0.5476|val 0.8339|Time 7.19mins
early stopping after epoch 019
RMSE: 0.8819
Save weight to:noSex_noAge_para3.weights
Prec10 0.7927|Rec10 0.5597
```

In [ ]:

parameter set 4

In [38]:
```
# model
n_factors = 150
hidden = [1000,1000,1000]
embedding_dropout = 0.05
dropouts = [0.5,0.5,0.25]
# training
lr = 1e-3
wd = 1e-5
bs =2000
n_epochs = 100
patience = 10

model = EmbeddingNet(n_users, n_movies,
                     n_factors=n_factors,
                     hidden=hidden,
                     embedding_dropout=embedding_dropout,
                     dropouts=dropouts)
best_weights = train_model(datasets, model, lr, wd, bs, n_epochs, patience)
df_final = get_result_df(datasets, model, best_weights, bs, save_path='noSex_noAge_para4.weights')
get_precision_recall(df_final, k=10, threshold=3.5)
```

```
loss improvement on epoch: 1
[001/100]|train 0.8841|val 0.8173|Time 0.70mins
loss improvement on epoch: 2
[002/100]|train 0.7926|val 0.7994|Time 1.39mins
loss improvement on epoch: 3
[003/100]|train 0.7956|val 0.7894|Time 2.11mins
loss improvement on epoch: 4
[004/100]|train 0.7514|val 0.7766|Time 2.96mins
[005/100]|train 0.7623|val 0.7789|Time 3.64mins
loss improvement on epoch: 6
[006/100]|train 0.7242|val 0.7737|Time 4.28mins
loss improvement on epoch: 7
[007/100]|train 0.7371|val 0.7736|Time 4.96mins
[008/100]|train 0.6834|val 0.7771|Time 5.64mins
[009/100]|train 0.6943|val 0.7759|Time 6.31mins
[010/100]|train 0.6254|val 0.7872|Time 6.99mins
[011/100]|train 0.6435|val 0.7832|Time 7.71mins
[012/100]|train 0.5742|val 0.8080|Time 8.44mins
[013/100]|train 0.5985|val 0.7931|Time 9.26mins
[014/100]|train 0.5342|val 0.8249|Time 9.96mins
[015/100]|train 0.5624|val 0.8118|Time 10.57mins
[016/100]|train 0.5047|val 0.8343|Time 11.23mins
[017/100]|train 0.5323|val 0.8220|Time 11.91mins
early stopping after epoch 017
RMSE: 0.8796
Save weight to:noSex_noAge_para4.weights
Prec10 0.7933|Rec10 0.5613
```

parameter set 5

In [39]:
```
# model --> seems that 100 is too small
n_factors = 100
hidden = [1000,1000,1000]
embedding_dropout = 0.05
dropouts = [0.5,0.5,0.25]
# training
lr = 1e-3
wd = 1e-5
bs =2000
n_epochs = 100
patience = 10

model = EmbeddingNet(n_users, n_movies,
                     n_factors=n_factors,
                     hidden=hidden,
                     embedding_dropout=embedding_dropout,
                     dropouts=dropouts)
best_weights = train_model(datasets, model, lr, wd, bs, n_epochs, patience)
df_final = get_result_df(datasets, model, best_weights, bs, save_path='noSex_noAge_para5.weights')
get_precision_recall(df_final, k=10, threshold=3.5)
```

```
loss improvement on epoch: 1
[001/100]|train 0.8860|val 0.8176|Time 0.64mins
loss improvement on epoch: 2
[002/100]|train 0.7949|val 0.8006|Time 1.27mins
loss improvement on epoch: 3
[003/100]|train 0.7972|val 0.7914|Time 2.07mins
loss improvement on epoch: 4
[004/100]|train 0.7550|val 0.7787|Time 2.86mins
[005/100]|train 0.7648|val 0.7788|Time 3.68mins
loss improvement on epoch: 6
[006/100]|train 0.7301|val 0.7747|Time 4.48mins
[007/100]|train 0.7428|val 0.7767|Time 5.19mins
[008/100]|train 0.6999|val 0.7791|Time 5.91mins
loss improvement on epoch: 9
[009/100]|train 0.7117|val 0.7738|Time 6.66mins
[010/100]|train 0.6538|val 0.7895|Time 7.38mins
[011/100]|train 0.6693|val 0.7781|Time 8.10mins
[012/100]|train 0.6076|val 0.8039|Time 8.83mins
[013/100]|train 0.6284|val 0.7898|Time 9.52mins
[014/100]|train 0.5683|val 0.8167|Time 10.27mins
[015/100]|train 0.5945|val 0.8026|Time 11.04mins
[016/100]|train 0.5389|val 0.8298|Time 11.73mins
[017/100]|train 0.5655|val 0.8138|Time 12.38mins
[018/100]|train 0.5145|val 0.8365|Time 13.01mins
[019/100]|train 0.5418|val 0.8313|Time 13.62mins
early stopping after epoch 019
RMSE: 0.8795
Save weight to:noSex_noAge_para5.weights
Prec10 0.7914|Rec10 0.5630
```

parameter set 6

In [40]:
```
# model
n_factors = 200
hidden = [1000,1000,1000]
embedding_dropout = 0.05
dropouts = [0.5,0.5,0.25]
# training
lr = 1e-3
wd = 1e-5
bs =2000
n_epochs = 100
patience = 10

model = EmbeddingNet(n_users, n_movies,
                     n_factors=n_factors,
                     hidden=hidden,
                     embedding_dropout=embedding_dropout,
                     dropouts=dropouts)
best_weights = train_model(datasets, model, lr, wd, bs, n_epochs, patience)
df_final = get_result_df(datasets, model, best_weights, bs, save_path='noSex_noAge_para6.weights')
get_precision_recall(df_final, k=10, threshold=3.5)
```

```
loss improvement on epoch: 1
[001/100]|train 0.8808|val 0.8184|Time 0.76mins
loss improvement on epoch: 2
[002/100]|train 0.7924|val 0.7982|Time 1.71mins
loss improvement on epoch: 3
[003/100]|train 0.7950|val 0.7894|Time 2.69mins
loss improvement on epoch: 4
[004/100]|train 0.7499|val 0.7760|Time 3.63mins
[005/100]|train 0.7619|val 0.7774|Time 4.55mins
loss improvement on epoch: 6
[006/100]|train 0.7226|val 0.7727|Time 5.50mins
[007/100]|train 0.7356|val 0.7733|Time 6.51mins
[008/100]|train 0.6779|val 0.7787|Time 7.52mins
loss improvement on epoch: 9
[009/100]|train 0.6890|val 0.7710|Time 8.47mins
[010/100]|train 0.6163|val 0.7891|Time 9.34mins
[011/100]|train 0.6359|val 0.7834|Time 10.28mins
[012/100]|train 0.5633|val 0.8084|Time 11.19mins
[013/100]|train 0.5883|val 0.7997|Time 12.06mins
[014/100]|train 0.5207|val 0.8225|Time 13.05mins
[015/100]|train 0.5497|val 0.8095|Time 14.01mins
[016/100]|train 0.4891|val 0.8394|Time 15.01mins
[017/100]|train 0.5185|val 0.8259|Time 16.57mins
[018/100]|train 0.4647|val 0.8504|Time 18.37mins
[019/100]|train 0.4941|val 0.8391|Time 20.25mins
early stopping after epoch 019
RMSE: 0.8787
Save weight to:noSex_noAge_para6.weights
Prec10 0.7968|Rec10 0.5528
```

parameter set 7

In [41]:
```
# model
n_factors = 250
hidden = [1000,1000,1000]
embedding_dropout = 0.05
dropouts = [0.5,0.5,0.25]
# training
lr = 1e-3
wd = 1e-5
bs =2000
n_epochs = 100
patience = 10

model = EmbeddingNet(n_users, n_movies,
                     n_factors=n_factors,
                     hidden=hidden,
                     embedding_dropout=embedding_dropout,
                     dropouts=dropouts)
best_weights = train_model(datasets, model, lr, wd, bs, n_epochs, patience)
df_final = get_result_df(datasets, model, best_weights, bs, save_path='noSex_noAge_para7.weights')
get_precision_recall(df_final, k=10, threshold=3.5)
```

```
loss improvement on epoch: 1
[001/100]|train 0.8802|val 0.8145|Time 1.90mins
loss improvement on epoch: 2
[002/100]|train 0.7910|val 0.7978|Time 3.50mins
loss improvement on epoch: 3
[003/100]|train 0.7936|val 0.7862|Time 5.18mins
loss improvement on epoch: 4
[004/100]|train 0.7475|val 0.7742|Time 6.75mins
[005/100]|train 0.7604|val 0.7780|Time 8.35mins
[006/100]|train 0.7212|val 0.7746|Time 9.92mins
[007/100]|train 0.7344|val 0.7754|Time 11.70mins
[008/100]|train 0.6760|val 0.7780|Time 13.28mins
[009/100]|train 0.6884|val 0.7756|Time 14.67mins
[010/100]|train 0.6115|val 0.7943|Time 15.47mins
[011/100]|train 0.6323|val 0.7851|Time 16.22mins
[012/100]|train 0.5540|val 0.8113|Time 16.98mins
[013/100]|train 0.5807|val 0.8003|Time 17.75mins
[014/100]|train 0.5111|val 0.8325|Time 18.49mins
early stopping after epoch 014
RMSE: 0.8800
Save weight to:noSex_noAge_para7.weights
Prec10 0.7922|Rec10 0.5558
```

In [ ]:

In [ ]:

In [ ]:

In [ ]: