ML-1M_dl_age

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]:
```
# load preprocessed df
df = pd.read_csv("ml-1m_dl.csv")
print(df.shape)
df.head()
```

```
(1000209, 9)
```

Out[3]:

	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 [4]:
```
# load dataset
datasets = pickle.load(open('ml-1m_dl.pkl','rb'))
datasets['val'][1]
```

Out[4]:
```
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
```

In [5]:
```
n_users = 6040
n_movies = 3706
dataset_sizes = {'train': 800167, 'val': 200042}
```

Define the network

In [6]:
```
# only consider sex
class EmbeddingNetAge(nn.Module):
    def __init__(self, n_users, n_movies, n_factors=50, embedding_dropout=0.02, hidden=10, dropouts=0.2, a_fa
ctor=25):
        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.g_factor = g_factor
        # self.g = nn.Embedding(2, self.g_factor)
        self.a_factor = a_factor
        self.a = nn.Embedding(7, self.a_factor)


        self.drop = nn.Dropout(embedding_dropout)
        self.hidden = nn.Sequential(*list(gen_layers(n_factors * 2 + self.a_factor)))
        self.fc = nn.Linear(n_last, 1)
        self._init()
    def forward(self, users, movies, ages, minmax=None):
        uu = self.u(users)
        # gg = self.g(genders)
        aa = self.a(ages)
        mm = self.m(movies)
        features = torch.cat([uu, aa, 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 [7]:
```python
# test
testnet = EmbeddingNetAge(n_users, n_movies, n_factors=150, hidden=100, dropouts=0.5, a_factor=25)
print(testnet)
testnet = EmbeddingNetAge(n_users, n_movies, n_factors=150, hidden=[100, 200, 300], dropouts=[0.25, 0.5], a_f
actor=25)
print(testnet)
```

```
EmbeddingNetAge(
  (u): Embedding(6041, 150)
  (m): Embedding(4000, 150)
  (a): Embedding(7, 25)
  (drop): Dropout(p=0.02, inplace=False)
  (hidden): Sequential(
    (0): Linear(in_features=325, out_features=100, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.5, inplace=False)
  )
  (fc): Linear(in_features=100, out_features=1, bias=True)
)
EmbeddingNetAge(
  (u): Embedding(6041, 150)
  (m): Embedding(4000, 150)
  (a): Embedding(7, 25)
  (drop): Dropout(p=0.02, inplace=False)
  (hidden): Sequential(
    (0): Linear(in_features=325, 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)
)
```

In [8]:
```python
# 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 [9]:
```
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 [10]:
```
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], x_batch[:, 3], 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 [11]:
```
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], x_batch[:, 3], 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], x_batch[:, 3], 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 [12]:
```
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))

# 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

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)
```

Now let's begin

Para 1

In [13]:
```
n_factors = 150
hidden = [500,500,500]
embedding_dropout = 0.05
dropouts = [0.5,0.5,0.25]
a_factor = 25

# training
lr = 1e-3
wd = 1e-5
bs =2000
n_epochs = 100
patience = 10


model = EmbeddingNetAge(n_users, n_movies,
                       n_factors=n_factors, hidden=hidden, dropouts=dropouts, a_factor=a_factor)
best_weights = train_model(datasets, model, lr, wd, bs, n_epochs, patience)
```

```
loss improvement on epoch: 1
[001/100]|train 0.9358|val 0.8268|Time 0.86mins
loss improvement on epoch: 2
[002/100]|train 0.8038|val 0.8069|Time 1.67mins
loss improvement on epoch: 3
[003/100]|train 0.8047|val 0.8012|Time 2.63mins
loss improvement on epoch: 4
[004/100]|train 0.7712|val 0.7918|Time 3.58mins
loss improvement on epoch: 5
[005/100]|train 0.7761|val 0.7820|Time 4.53mins
loss improvement on epoch: 6
[006/100]|train 0.7387|val 0.7735|Time 5.45mins
loss improvement on epoch: 7
[007/100]|train 0.7480|val 0.7734|Time 6.31mins
loss improvement on epoch: 8
[008/100]|train 0.7140|val 0.7659|Time 7.22mins
[009/100]|train 0.7269|val 0.7672|Time 8.04mins
loss improvement on epoch: 10
[010/100]|train 0.6953|val 0.7654|Time 8.87mins
loss improvement on epoch: 11
[011/100]|train 0.7111|val 0.7642|Time 9.70mins
[012/100]|train 0.6795|val 0.7660|Time 10.53mins
loss improvement on epoch: 13
[013/100]|train 0.6956|val 0.7640|Time 11.32mins
[014/100]|train 0.6636|val 0.7664|Time 12.12mins
[015/100]|train 0.6802|val 0.7648|Time 13.03mins
[016/100]|train 0.6476|val 0.7692|Time 13.93mins
[017/100]|train 0.6649|val 0.7693|Time 14.80mins
[018/100]|train 0.6316|val 0.7753|Time 15.69mins
[019/100]|train 0.6504|val 0.7710|Time 16.48mins
[020/100]|train 0.6157|val 0.7794|Time 17.33mins
[021/100]|train 0.6347|val 0.7779|Time 18.21mins
[022/100]|train 0.6014|val 0.7852|Time 19.04mins
[023/100]|train 0.6207|val 0.7801|Time 19.85mins
early stopping after epoch 023
```

In [14]:
```
df_final = get_result_df(datasets, model, best_weights, bs, save_path='noSex_WithAge_para1.weights')
get_precision_recall(df_final, k=10, threshold=3.5)
```

```
RMSE: 0.8736
Save weight to:noSex_WithAge_para1.weights
Prec10 0.8003|Rec10 0.5586
```

para2

In [15]:
```
n_factors = 150
hidden = [500,500,500]
embedding_dropout = 0.05
dropouts = [0.5,0.5,0.25]
a_factor = 50

# training
lr = 1e-3
wd = 1e-5
bs =2000
n_epochs = 100
patience = 10


model = EmbeddingNetAge(n_users, n_movies,
                       n_factors=n_factors, hidden=hidden, dropouts=dropouts, a_factor=a_factor)
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_WithAge_para2.weights')
get_precision_recall(df_final, k=10, threshold=3.5)
```

```
loss improvement on epoch: 1
[001/100]|train 0.9626|val 0.8315|Time 0.43mins
loss improvement on epoch: 2
[002/100]|train 0.8085|val 0.8106|Time 0.83mins
loss improvement on epoch: 3
[003/100]|train 0.8087|val 0.8053|Time 1.27mins
loss improvement on epoch: 4
[004/100]|train 0.7779|val 0.7962|Time 1.70mins
loss improvement on epoch: 5
[005/100]|train 0.7852|val 0.7921|Time 2.12mins
loss improvement on epoch: 6
[006/100]|train 0.7531|val 0.7827|Time 2.52mins
loss improvement on epoch: 7
[007/100]|train 0.7601|val 0.7783|Time 2.93mins
loss improvement on epoch: 8
[008/100]|train 0.7267|val 0.7711|Time 3.33mins
loss improvement on epoch: 9
[009/100]|train 0.7381|val 0.7697|Time 3.76mins
loss improvement on epoch: 10
[010/100]|train 0.7082|val 0.7671|Time 4.19mins
loss improvement on epoch: 11
[011/100]|train 0.7226|val 0.7661|Time 4.61mins
[012/100]|train 0.6939|val 0.7663|Time 5.03mins
loss improvement on epoch: 13
[013/100]|train 0.7094|val 0.7659|Time 5.43mins
loss improvement on epoch: 14
[014/100]|train 0.6809|val 0.7656|Time 5.85mins
loss improvement on epoch: 15
[015/100]|train 0.6968|val 0.7647|Time 6.24mins
[016/100]|train 0.6677|val 0.7667|Time 6.65mins
[017/100]|train 0.6844|val 0.7657|Time 7.07mins
[018/100]|train 0.6554|val 0.7679|Time 7.49mins
[019/100]|train 0.6721|val 0.7695|Time 7.88mins
[020/100]|train 0.6419|val 0.7727|Time 8.29mins
[021/100]|train 0.6602|val 0.7674|Time 8.69mins
[022/100]|train 0.6302|val 0.7770|Time 9.08mins
[023/100]|train 0.6483|val 0.7742|Time 9.49mins
[024/100]|train 0.6179|val 0.7775|Time 9.87mins
[025/100]|train 0.6373|val 0.7803|Time 10.24mins
early stopping after epoch 025
RMSE: 0.8743
Save weight to:noSex_WithAge_para2.weights
Prec10 0.7987|Rec10 0.5625
```

para3

In [16]:
```
n_factors = 200
hidden = [500,500,500]
embedding_dropout = 0.05
dropouts = [0.5,0.5,0.25]
a_factor = 25

# training
lr = 1e-3
wd = 1e-5
bs =2000
n_epochs = 100
patience = 10


model = EmbeddingNetAge(n_users, n_movies,
                       n_factors=n_factors, hidden=hidden, dropouts=dropouts, a_factor=a_factor)
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_WithAge_para3.weights')
get_precision_recall(df_final, k=10, threshold=3.5)
```

```
loss improvement on epoch: 1
[001/100]|train 0.9363|val 0.8253|Time 0.42mins
loss improvement on epoch: 2
[002/100]|train 0.8029|val 0.8064|Time 0.90mins
loss improvement on epoch: 3
[003/100]|train 0.8043|val 0.8016|Time 1.30mins
loss improvement on epoch: 4
[004/100]|train 0.7698|val 0.7900|Time 1.78mins
loss improvement on epoch: 5
[005/100]|train 0.7734|val 0.7814|Time 2.34mins
loss improvement on epoch: 6
[006/100]|train 0.7354|val 0.7721|Time 2.86mins
loss improvement on epoch: 7
[007/100]|train 0.7449|val 0.7683|Time 3.38mins
loss improvement on epoch: 8
[008/100]|train 0.7097|val 0.7657|Time 3.90mins
[009/100]|train 0.7239|val 0.7663|Time 4.47mins
loss improvement on epoch: 10
[010/100]|train 0.6896|val 0.7654|Time 5.01mins
loss improvement on epoch: 11
[011/100]|train 0.7055|val 0.7645|Time 5.52mins
loss improvement on epoch: 12
[012/100]|train 0.6715|val 0.7641|Time 6.09mins
loss improvement on epoch: 13
[013/100]|train 0.6886|val 0.7641|Time 6.64mins
[014/100]|train 0.6535|val 0.7667|Time 7.16mins
[015/100]|train 0.6712|val 0.7674|Time 7.69mins
[016/100]|train 0.6360|val 0.7699|Time 8.16mins
[017/100]|train 0.6551|val 0.7684|Time 8.59mins
[018/100]|train 0.6184|val 0.7766|Time 9.02mins
[019/100]|train 0.6386|val 0.7736|Time 9.44mins
[020/100]|train 0.6021|val 0.7816|Time 9.88mins
[021/100]|train 0.6220|val 0.7845|Time 10.33mins
[022/100]|train 0.5857|val 0.7876|Time 10.76mins
[023/100]|train 0.6064|val 0.7833|Time 11.20mins
early stopping after epoch 023
RMSE: 0.8734
Save weight to:noSex_WithAge_para3.weights
Prec10 0.8002|Rec10 0.5685
```

para4

In [17]:
```
n_factors = 200
hidden = [500,500,500]
embedding_dropout = 0.05
dropouts = [0.5,0.5,0.25]
a_factor = 50

# training
lr = 1e-3
wd = 1e-5
bs =2000
n_epochs = 100
patience = 10


model = EmbeddingNetAge(n_users, n_movies,
                       n_factors=n_factors, hidden=hidden, dropouts=dropouts, a_factor=a_factor)
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_WithAge_para4.weights')
get_precision_recall(df_final, k=10, threshold=3.5)
```

```
loss improvement on epoch: 1
[001/100]|train 0.9533|val 0.8300|Time 0.43mins
loss improvement on epoch: 2
[002/100]|train 0.8066|val 0.8092|Time 0.89mins
loss improvement on epoch: 3
[003/100]|train 0.8079|val 0.8044|Time 1.30mins
loss improvement on epoch: 4
[004/100]|train 0.7755|val 0.7949|Time 1.72mins
loss improvement on epoch: 5
[005/100]|train 0.7822|val 0.7887|Time 2.11mins
loss improvement on epoch: 6
[006/100]|train 0.7473|val 0.7780|Time 2.48mins
loss improvement on epoch: 7
[007/100]|train 0.7554|val 0.7735|Time 2.93mins
loss improvement on epoch: 8
[008/100]|train 0.7219|val 0.7685|Time 3.38mins
[009/100]|train 0.7352|val 0.7699|Time 3.81mins
loss improvement on epoch: 10
[010/100]|train 0.7042|val 0.7668|Time 4.23mins
[011/100]|train 0.7189|val 0.7671|Time 4.71mins
[012/100]|train 0.6899|val 0.7669|Time 5.14mins
loss improvement on epoch: 13
[013/100]|train 0.7053|val 0.7660|Time 5.58mins
[014/100]|train 0.6757|val 0.7695|Time 6.04mins
[015/100]|train 0.6922|val 0.7695|Time 6.51mins
[016/100]|train 0.6610|val 0.7709|Time 7.01mins
[017/100]|train 0.6792|val 0.7696|Time 7.47mins
[018/100]|train 0.6478|val 0.7749|Time 7.97mins
[019/100]|train 0.6663|val 0.7686|Time 8.42mins
[020/100]|train 0.6347|val 0.7769|Time 8.82mins
[021/100]|train 0.6533|val 0.7722|Time 9.22mins
[022/100]|train 0.6211|val 0.7842|Time 9.66mins
[023/100]|train 0.6403|val 0.7775|Time 10.09mins
early stopping after epoch 023
RMSE: 0.8749
Save weight to:noSex_WithAge_para4.weights
Prec10 0.7971|Rec10 0.5643
```

In [ ]:

In [ ]: