Lecture 07: Collaborative filtering class demo

Contents

```
import os
import random
import sys

import numpy as np
import pandas as pd

DATA_DIR = os.path.join(os.path.abspath(".."), "data/")

sys.path.append(os.path.join(os.path.abspath(".."), "code"))
```

Example dataset: Jester 1.7M jokes ratings dataset

- We'll use a sample of Jester 1.7M jokes ratings dataset to demonstrate different recommendation systems.

The dataset comes with two CSVs

- A CSV containing ratings (-10.0 to +10.0) of 150 jokes from 59,132 users.
- A CSV containing joke IDs and the actual text of jokes.

> Some jokes might be offensive. Please do not look too much into the actual text data if you are sensitive to such language.

- Recommendation systems are most effective when you have a large amount of data.
- But we are only taking a sample here for speed.

```
filename = DATA_DIR + "jester_ratings.csv"
ratings_full = pd.read_csv(filename)
ratings = ratings_full[ratings_full["userId"] <= 4000]
```

```
ratings.head()
```

	userId	jokeId	rating
0	1	5	0.219
1	1	7	-9.281
2	1	8	-9.281
3	1	13	-6.781
4	1	15	0.875

```
user_key = "userId"
item_key = "jokeId"
```

Dataset stats

```
ratings.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 141362 entries, 0 to 141361
Data columns (total 3 columns):
 #   Column  Non-Null Count   Dtype  
---  ------  --------------   -----  
 0   userId  141362 non-null  int64  
 1   jokeId  141362 non-null  int64  
 2   rating  141362 non-null  float64
dtypes: float64(1), int64(2)
memory usage: 4.3 MB
```

```
def get_stats(ratings, item_key="jokeId", user_key="userId"):
    print("Number of ratings:", len(ratings))
    print("Average rating:  %0.3f" % (np.mean(ratings["rating"])))
    N = len(np.unique(ratings[user_key]))
    M = len(np.unique(ratings[item_key]))
    print("Number of users (N): %d" % N)
    print("Number of items (M): %d" % M)
    print("Fraction non-nan ratings: %0.3f" % (len(ratings) / (N * M)))
    return N, M


N, M = get_stats(ratings)
```

```
Number of ratings: 141362
Average rating:  1.200
Number of users (N): 3635
Number of items (M): 140
Fraction non-nan ratings: 0.278
```

- Let's construct utility matrix with `number of users` rows and `number of items` columns from the ratings data.

> Note we are constructing a non-sparse matrix for demonstration purpose here. In real life it's recommended that you work with sparse matrices.

```
user_mapper = dict(zip(np.unique(ratings[user_key]), list(range(N))))
item_mapper = dict(zip(np.unique(ratings[item_key]), list(range(M))))
user_inverse_mapper = dict(zip(list(range(N)), np.unique(ratings[user_key])))
item_inverse_mapper = dict(zip(list(range(M)), np.unique(ratings[item_key])))
```

```
def create_Y_from_ratings(
    data, N, M, user_mapper, item_mapper, user_key="userId", item_key="jokeId"
):  # Function to create a dense utility matrix
    Y = np.zeros((N, M))
    Y.fill(np.nan)
    for index, val in data.iterrows():
        n = user_mapper[val[user_key]]
        m = item_mapper[val[item_key]]
        Y[n, m] = val["rating"]

    return Y
```

Utility matrix for Jester jokes ratings data

- Rows represent users.
- Columns represent items (jokes in our case).
- Each cell gives the rating given by the user to the corresponding joke.
- Users are features for jokes and jokes are features for users.
- We want to predict the missing entries.

```
Y_mat = create_Y_from_ratings(ratings, N, M, user_mapper, item_mapper)
Y_mat.shape
```

```
(3635, 140)
```

```
pd.DataFrame(Y_mat)
```

	0	1	2	3	4	5	6	7	8	
0	0.219	-9.281	-9.281	-6.781	0.875	-9.656	-9.031	-7.469	-8.719	-
1	-9.688	9.938	9.531	9.938	0.406	3.719	9.656	-2.688	-9.562	-
2	-9.844	-9.844	-7.219	-2.031	-9.938	-9.969	-9.875	-9.812	-9.781	-
3	-5.812	-4.500	-4.906	NaN	NaN	NaN	NaN	NaN	NaN	
4	6.906	4.750	-5.906	-0.406	-4.031	3.875	6.219	5.656	6.094	
...	...	...	...	...	...	...	...	...	...	
3630	NaN	-9.812	-0.062	NaN	NaN	NaN	NaN	NaN	NaN	
3631	NaN	-9.844	7.531	-9.719	-9.344	3.875	9.812	8.938	8.375	
3632	NaN	-1.906	3.969	-2.312	-0.344	-8.844	4.188	NaN	NaN	
3633	NaN	-8.875	-9.156	-9.156	NaN	NaN	NaN	NaN	NaN	
3634	NaN	-6.312	1.281	-3.531	2.125	-5.812	5.562	-6.062	0.125	

3635 rows × 140 columns

Data splitting and evaluation

- Recall that our goal is to predict missing entries in the utility matrix.

```
pd.DataFrame(Y_mat)
```

	0	1	2	3	4	5	6	7	8	
0	0.219	-9.281	-9.281	-6.781	0.875	-9.656	-9.031	-7.469	-8.719	-
1	-9.688	9.938	9.531	9.938	0.406	3.719	9.656	-2.688	-9.562	-
2	-9.844	-9.844	-7.219	-2.031	-9.938	-9.969	-9.875	-9.812	-9.781	-
3	-5.812	-4.500	-4.906	NaN	NaN	NaN	NaN	NaN	NaN	
4	6.906	4.750	-5.906	-0.406	-4.031	3.875	6.219	5.656	6.094	
...	...	...	...	...	...	...	...	...	...	
3630	NaN	-9.812	-0.062	NaN	NaN	NaN	NaN	NaN	NaN	
3631	NaN	-9.844	7.531	-9.719	-9.344	3.875	9.812	8.938	8.375	
3632	NaN	-1.906	3.969	-2.312	-0.344	-8.844	4.188	NaN	NaN	
3633	NaN	-8.875	-9.156	-9.156	NaN	NaN	NaN	NaN	NaN	
3634	NaN	-6.312	1.281	-3.531	2.125	-5.812	5.562	-6.062	0.125	

3635 rows × 140 columns

Data splitting

- We split the ratings into train and validation sets.
- It's easier to split the ratings data instead of splitting the utility matrix.
- Don't worry about `y`; we're not really going to use it.

```
from sklearn.model_selection import train_test_split
X = ratings.copy()
y = ratings[user_key]
X_train, X_valid, y_train, y_valid = train_test_split(
    X, y, test_size=0.2, random_state=42
)

X_train.shape, X_valid.shape
```

```
((113089, 3), (28273, 3))
```

Now we will create utility matrices for train and validation splits.

```
train_mat = create_Y_from_ratings(X_train, N, M, user_mapper, item_mapper)
valid_mat = create_Y_from_ratings(X_valid, N, M, user_mapper, item_mapper)
```

```
train_mat.shape, valid_mat.shape
```

```
((3635, 140), (3635, 140))
```

```
(len(X_train) / (N * M)) # Fraction of non-nan entries in the train set
```

```
0.22222244055806642
```

```
(len(X_valid) / (N * M)) # Fraction of non-nan entries in the valid set
```

```
0.055557083906464924
```

- `train_mat` has only ratings from the train set and `valid_mat` has only ratings from the valid set.
- During training we assume that we do not have access to some of the available ratings. We predict these ratings and evaluate them against ratings in the validation set.

```
def error(X1, X2):
    """
    Returns the root mean squared error.
    """
    return np.sqrt(np.nanmean((X1 - X2) ** 2))


def evaluate(pred_X, train_X, valid_X, model_name="Global average"):
    print("%s train RMSE: %0.2f" % (model_name, error(pred_X, train_X)))
    print("%s valid RMSE: %0.2f" % (model_name, error(pred_X, valid_X)))
```

Baseline approaches

Let's first try some simple approaches to predict missing entries.

1. Global average baseline
2. Per-user average baseline
3. Per-item average baseline
4. Average of 2 and 3
    - Take an average of per-user and per-item averages.
5. k-Nearest Neighbours imputation

I'll show you 1. and 5. You'll explore 2., 3., and 4. in the lab.

Global average baseline

- Let's examine RMSE of the global average baseline.
- In this baseline we predict everything as the global average rating.

```
avg = np.nanmean(train_mat)
pred_g = np.zeros(train_mat.shape) + avg
pd.DataFrame(pred_g).head()
```

	0	1	2	3	4	5	6	7	8
0	1.20741	1.20741	1.20741	1.20741	1.20741	1.20741	1.20741	1.20741	1.20741
1	1.20741	1.20741	1.20741	1.20741	1.20741	1.20741	1.20741	1.20741	1.20741
2	1.20741	1.20741	1.20741	1.20741	1.20741	1.20741	1.20741	1.20741	1.20741
3	1.20741	1.20741	1.20741	1.20741	1.20741	1.20741	1.20741	1.20741	1.20741
4	1.20741	1.20741	1.20741	1.20741	1.20741	1.20741	1.20741	1.20741	1.20741

5 rows × 140 columns

```
evaluate(pred_g, train_mat, valid_mat, model_name="Global average")
```

```
Global average train RMSE: 5.75
Global average valid RMSE: 5.77
```

k-nearest neighbours imputation

```
from sklearn.impute import KNNImputer

imputer = KNNImputer(n_neighbors=10, keep_empty_features=True)
train_mat_imp = imputer.fit_transform(train_mat)
```

```
pd.DataFrame(train_mat_imp)
```

	0	1	2	3	4	5	6	7
0	-5.9406	-9.2810	-9.2810	-6.7810	0.8750	-9.6560	-9.0310	-7.4690
1	2.3405	9.9380	9.5310	9.9380	0.4060	3.7190	9.6560	-2.6880
2	-9.8440	-3.5750	-7.2190	-2.0310	-9.9380	-9.9690	-9.8750	-9.8120
3	-5.8120	-2.4624	-4.9060	-2.7781	-0.0532	-3.8594	1.7031	-0.3687
4	1.3157	4.7500	1.8658	-0.4060	1.7937	3.8750	6.2190	1.9220
...	...	...	...	...	...	...	...	...
3630	-0.7750	-9.8120	-0.0620	-2.8218	-3.5282	-4.8281	2.2718	-2.8782
3631	2.5188	-5.0625	-0.4001	-9.7190	-9.3440	-1.6408	-4.1187	8.9380
3632	0.1749	-1.9060	3.9690	-1.3844	-0.3440	-8.8440	4.1880	-1.5564
3633	-2.7749	-6.4907	-6.1594	-9.1560	-7.1437	-6.5406	3.8718	-1.7782
3634	-0.0812	-6.3120	1.2810	-3.5310	2.1250	-5.8120	5.5620	0.2218

3635 rows × 140 columns

```
evaluate(train_mat_imp, train_mat, valid_mat, model_name="KNN imputer")
```

```
KNN imputer train RMSE: 0.00
KNN imputer valid RMSE: 4.79
```

(Optional) Finding nearest neighbors

- We can look at nearest neighbours of a query item.
- Here our columns are jokes, and users are features for jokes, and we'll have to find nearest neighbours of columns vectors.

```
pd.DataFrame(train_mat_imp).head()
```

	0	1	2	3	4	5	6	7	
0	-5.9406	-9.2810	-9.2810	-6.7810	0.8750	-9.6560	-9.0310	-7.4690	-8.7
1	2.3405	9.9380	9.5310	9.9380	0.4060	3.7190	9.6560	-2.6880	4.3
2	-9.8440	-3.5750	-7.2190	-2.0310	-9.9380	-9.9690	-9.8750	-9.8120	-9.7
3	-5.8120	-2.4624	-4.9060	-2.7781	-0.0532	-3.8594	1.7031	-0.3687	1.8
4	1.3157	4.7500	1.8658	-0.4060	1.7937	3.8750	6.2190	1.9220	6.0

5 rows × 140 columns

(Optional) k-nearest neighbours on a query joke

- Let's transpose the matrix.

```
item_user_mat = train_mat_imp.T
```

```
jokes_df = pd.read_csv("../data/jester_items.csv")
jokes_df.head()
```

	jokeId	jokeText
0	1	A man visits the doctor. The doctor says "I ha...
1	2	This couple had an excellent relationship goin...
2	3	Q. What's 200 feet long and has 4 teeth? \n\nA...
3	4	Q. What's the difference between a man and a t...
4	5	Q.\tWhat's O. J. Simpson's Internet address? \...

```
jester_items_df = pd.read_csv("../data/jester_items.csv")
jester_items_df.head()
```

	jokeId	jokeText
0	1	A man visits the doctor. The doctor says "I ha...
1	2	This couple had an excellent relationship goin...
2	3	Q. What's 200 feet long and has 4 teeth? \n\nA...
3	4	Q. What's the difference between a man and a t...
4	5	Q.\tWhat's O. J. Simpson's Internet address? \...

```
id_joke_map = dict(zip(jokes_df.jokeId, jokes_df.jokeText))
```

```
from sklearn.neighbors import NearestNeighbors


def get_topk_recommendations(X, query_ind=0, metric="cosine", k=5):
    query_idx = item_inverse_mapper[query_ind]
    model = NearestNeighbors(n_neighbors=k, metric="cosine")
    model.fit(X)
    neigh_ind = model.kneighbors([X[query_ind]], k, return_distance=False).fla
    neigh_ind = np.delete(neigh_ind, np.where(query_ind == query_ind))
    recs = [id_joke_map[item_inverse_mapper[i]] for i in neigh_ind]
    print("Query joke: ", id_joke_map[query_idx])

    return pd.DataFrame(data=recs, columns=["top recommendations"])


get_topk_recommendations(item_user_mat, query_ind=8, metric="cosine", k=5)
```

```
Query joke:  Q: If a person who speaks three languages is called "tri-lingual,"
a person who speaks two languages is called "bi-lingual," what do call
a person who only speaks one language?

A: American!
```

```
/var/folders/b3/g26r0dcx4b35vf3nk31216hc0000gr/T/ipykernel_18661/3461118492.py
  neigh_ind = np.delete(neigh_ind, np.where(query_ind == query_ind))
```

	top recommendations
0	Q: What is the difference between George Wash...
1	A man in a hot air balloon realized he was los...
2	If pro- is the opposite of con- then congress ...
3	Arnold Swartzeneger and Sylvester Stallone are...

Collaborative filtering using the `surprise` package

Although matrix factorization is a prominent approach to complete the utility matrix, `TruncatedSVD` is not appropriate in this context because of a large number of NaN values in this matrix.

- We consider only observed ratings and add regularization to avoid overfitting.
- Here is the loss function

$$f(Z, W) = \sum_{(i,j) \in R} ((w_j^T z_i) - y_{ij})^2 + \frac{\lambda_1}{2} \|Z\|_2^2 + \frac{\lambda_2}{2} \|W\|_2^2$$

Let's try it out on our Jester dataset utility matrix.

```
import surprise
from surprise import SVD, Dataset, Reader, accuracy
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[29], line 1
----> 1 import surprise
      2 from surprise import SVD, Dataset, Reader, accuracy

ModuleNotFoundError: No module named 'surprise'
```

```
reader = Reader()
data = Dataset.load_from_df(ratings, reader)  # Load the data

# I'm being sloppy here. Probably there is a way to create validset from our a
trainset, validset = surprise.model_selection.train_test_split(
    data, test_size=0.2, random_state=42
)  # Split the data
```

Regularized SVD

```
k = 10
algo = SVD(n_factors=k, random_state=42)
algo.fit(trainset)
svd_preds = algo.test(validset)
accuracy.rmse(svd_preds, verbose=True)
```

```
RMSE: 5.2893
```

```
5.28926338380112
```

- No big improvement over the global baseline (RMSE=5.77).
- Probably because we are only considering a sample.

Cross-validation for recommender systems

- We can also carry out cross-validation and grid search with this package.
- Let's look at an example of cross-validation.

```
from surprise.model_selection import cross_validate

pd.DataFrame(cross_validate(algo, data, measures=["RMSE", "MAE"], cv=5, verbos
```

```
Evaluating RMSE, MAE of algorithm SVD on 5 split(s).

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
RMSE (testset)    5.2773  5.2694  5.2875  5.2956  5.3009  5.2862  0.0116
MAE (testset)     4.1974  4.1862  4.2059  4.2035  4.2126  4.2011  0.0089
Fit time          0.16    0.16    0.16    0.20    0.19    0.17    0.02
Test time         0.10    0.10    0.05    0.11    0.05    0.08    0.02
```

	test_rmse	test_mae	fit_time	test_time
0	5.277348	4.197364	0.161883	0.095874
1	5.269368	4.186245	0.157400	0.096561
2	5.287539	4.205930	0.156934	0.053814
3	5.295635	4.203529	0.199135	0.112551
4	5.300932	4.212576	0.187679	0.052672

- Jester dataset is available as one of the built-in datasets in this package and you can load it as follows and run cross-validation as follows.

```
data = Dataset.load_builtin("jester")

pd.DataFrame(cross_validate(algo, data, measures=["RMSE", "MAE"], cv=5, verbos
```

```
Evaluating RMSE, MAE of algorithm SVD on 5 split(s).

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
RMSE (testset)    4.5986  4.6054  4.5847  4.5973  4.5770  4.5926  0.0103
MAE (testset)     3.3261  3.3389  3.3199  3.3346  3.3235  3.3286  0.0071
Fit time          2.83    3.38    2.95    2.95    2.80    2.98    0.21
Test time         2.01    2.01    1.69    1.71    1.73    1.83    0.15
```

	test_rmse	test_mae	fit_time	test_time
0	4.598583	3.326069	2.827352	2.008931
1	4.605385	3.338924	3.377606	2.010812
2	4.584676	3.319896	2.945285	1.693255
3	4.597327	3.334622	2.953656	1.706650
4	4.577031	3.323497	2.802552	1.725876

(Optional) PyTorch implementation

We can also implement the loss function above using `PyTorch`.

```
import pandas as pd

# Load the dataset
ratings = pd.read_csv('../data/jester_ratings.csv')
print(ratings.head())
```

```
   userId  jokeId  rating
0       1       5   0.219
1       1       7  -9.281
2       1       8  -9.281
3       1      13  -6.781
4       1      15   0.875
```

```
ratings_df = pd.read_csv('../data/jester_ratings.csv')
ratings = ratings_df[ratings_df["userId"] <= 4000].copy()
# ratings = ratings_full[ratings_full["userId"] <= 4000]
```

```
ratings
```

	userId	jokeId	rating
0	1	5	0.219
1	1	7	-9.281
2	1	8	-9.281
3	1	13	-6.781
4	1	15	0.875
...	...	...	...
141357	4000	18	-6.062
141358	4000	19	0.125
141359	4000	76	5.719
141360	4000	53	5.344
141361	4000	143	4.188

141362 rows × 3 columns

```
from sklearn.preprocessing import LabelEncoder

user_encoder = LabelEncoder()
ratings['user'] = user_encoder.fit_transform(ratings.userId.values)

item_encoder = LabelEncoder()
ratings['item'] = item_encoder.fit_transform(ratings.jokeId.values)

num_users = ratings['user'].nunique()
num_items = ratings['item'].nunique()
```

```
ratings
```

	userId	jokeId	rating	user	item
0	1	5	0.219	0	0
1	1	7	-9.281	0	1
2	1	8	-9.281	0	2
3	1	13	-6.781	0	3
4	1	15	0.875	0	4
...	...	...	...	...	...
141357	4000	18	-6.062	3634	7
141358	4000	19	0.125	3634	8
141359	4000	76	5.719	3634	65
141360	4000	53	5.344	3634	42
141361	4000	143	4.188	3634	132

141362 rows × 5 columns

```
X = ratings.copy()
y = ratings[user_key]
X_train, X_valid, y_train, y_valid = train_test_split(
    X, y, test_size=0.2, random_state=42
)

X_train.shape, X_valid.shape
```

```
((113089, 5), (28273, 5))
```

Let's create a custom `ItemsRatingsDataset` class for the ratings data, so that we can use of PyTorch's DataLoader for batch processing and data shuffling during training.

```
import torch
from torch.utils.data import Dataset, DataLoader

class ItemsRatingsDataset(Dataset):
    def __init__(self, users, items, ratings):
        self.users = users
        self.items = items
        self.ratings = ratings

    def __len__(self):
        return len(self.ratings)

    def __getitem__(self, idx):
        return {
            'user': torch.tensor(self.users[idx], dtype=torch.long),
            'item': torch.tensor(self.items[idx], dtype=torch.long),
            'rating': torch.tensor(self.ratings[idx], dtype=torch.float)
        }

train_dataset = ItemsRatingsDataset(X_train['user'].values, X_train['item'].va
train_dataloader = DataLoader(train_dataset, batch_size=512, shuffle=True)

valid_dataset = ItemsRatingsDataset(X_valid['user'].values, X_valid['item'].va
valid_dataloader = DataLoader(valid_dataset, batch_size=512, shuffle=True)
```

The `CFModel` class below defines the architecture and the forward method of collaborative filtering. We are using the `embedding` layer of `torch.nn` which simply creates a lookup table that stores embeddings of a fixed dictionary and size.

```
import torch.nn as nn

class CFModel(nn.Module):
    def __init__(self, num_users, num_items, emb_size=100):
        super(CFModel, self).__init__()
        # Embeddings for users
        self.user_emb = nn.Embedding(num_users, emb_size)

        # Embeddings for items
        self.item_emb = nn.Embedding(num_items, emb_size)

        # Weight initialization
        nn.init.xavier_uniform_(self.user_emb.weight)
        nn.init.xavier_uniform_(self.item_emb.weight)

    def forward(self, user, movie):
        user_embedding = self.user_emb(user)
        item_embedding = self.item_emb(movie)
        # calculate predicted ratings as dot products
        # of corresponding user and item embeddings
        return (user_embedding * item_embedding).sum(1)

model = CFModel(num_users, num_items)
```

```
import torch.optim as optim

# Loss function
criterion = nn.MSELoss()

# Regularization coefficient
lambda_reg = 0.001

# Optimizer (without weight decay)
# We manually add regularization in the loss below
optimizer = optim.Adam(model.parameters(), lr=0.001)


import torch
import numpy as np
import torch.nn.functional as F

def train(model, train_dataloader, valid_dataloader, optimizer, criterion, epo
    for epoch in range(epochs):
        model.train()
        total_train_loss = 0
        for batch in train_dataloader:
            optimizer.zero_grad()
            predictions = model(batch['user'], batch['item'])
            loss = criterion(predictions, batch['rating'])

            # Normalize regularization terms by batch size
            batch_size = batch['user'].size(0)
            user_reg = lambda_reg * model.user_emb.weight.norm(2) / batch_size
            item_reg = lambda_reg * model.item_emb.weight.norm(2) / batch_size

            train_loss = loss + user_reg + item_reg
            train_loss.backward()
            optimizer.step()
            total_train_loss += train_loss.item()

        avg_train_loss = total_train_loss / len(train_dataloader)

        model.eval()
        total_valid_loss = 0
        all_predictions = []
        all_ratings = []
        with torch.no_grad():
            for batch in valid_dataloader:
                predictions = model(batch['user'], batch['item'])
                valid_loss = criterion(predictions, batch['rating'])
                total_valid_loss += valid_loss.item()
                all_predictions.extend(predictions.detach().cpu().tolist())
                all_ratings.extend(batch['rating'].cpu().tolist())

        avg_valid_loss = total_valid_loss / len(valid_dataloader)
        rmse = np.sqrt(np.mean((np.array(all_predictions) - np.array(all_ratin
        print(f'Epoch {epoch+1}, Train Loss: {avg_train_loss:.4f}, Valid Loss:
```

```
train(model, train_dataloader, valid_dataloader, optimizer, criterion, epochs
```

```
Epoch 1, Train Loss: 34.4687, Valid Loss: 34.2289, Valid RMSE: 5.8551
Epoch 2, Train Loss: 31.7923, Valid Loss: 28.6969, Valid RMSE: 5.3602
Epoch 3, Train Loss: 24.3265, Valid Loss: 23.4747, Valid RMSE: 4.8476
Epoch 4, Train Loss: 20.3614, Valid Loss: 22.0125, Valid RMSE: 4.6892
Epoch 5, Train Loss: 18.3417, Valid Loss: 21.2944, Valid RMSE: 4.6168
Epoch 6, Train Loss: 16.9408, Valid Loss: 20.9704, Valid RMSE: 4.5796
Epoch 7, Train Loss: 15.7807, Valid Loss: 20.7384, Valid RMSE: 4.5573
Epoch 8, Train Loss: 14.7120, Valid Loss: 20.6459, Valid RMSE: 4.5422
Epoch 9, Train Loss: 13.7044, Valid Loss: 20.5807, Valid RMSE: 4.5320
Epoch 10, Train Loss: 12.7627, Valid Loss: 20.4953, Valid RMSE: 4.5250
Epoch 11, Train Loss: 11.8992, Valid Loss: 20.3900, Valid RMSE: 4.5215
Epoch 12, Train Loss: 11.1100, Valid Loss: 20.4123, Valid RMSE: 4.5207
Epoch 13, Train Loss: 10.3811, Valid Loss: 20.5020, Valid RMSE: 4.5224
Epoch 14, Train Loss: 9.7040, Valid Loss: 20.4828, Valid RMSE: 4.5266
Epoch 15, Train Loss: 9.0698, Valid Loss: 20.5114, Valid RMSE: 4.5331
Epoch 16, Train Loss: 8.4661, Valid Loss: 20.5815, Valid RMSE: 4.5406
Epoch 17, Train Loss: 7.8923, Valid Loss: 20.6766, Valid RMSE: 4.5501
Epoch 18, Train Loss: 7.3442, Valid Loss: 20.7467, Valid RMSE: 4.5610
Epoch 19, Train Loss: 6.8243, Valid Loss: 20.9451, Valid RMSE: 4.5738
Epoch 20, Train Loss: 6.3301, Valid Loss: 21.0926, Valid RMSE: 4.5874
Epoch 21, Train Loss: 5.8663, Valid Loss: 21.1772, Valid RMSE: 4.6021
Epoch 22, Train Loss: 5.4281, Valid Loss: 21.3600, Valid RMSE: 4.6171
Epoch 23, Train Loss: 5.0210, Valid Loss: 21.4685, Valid RMSE: 4.6339
Epoch 24, Train Loss: 4.6420, Valid Loss: 21.6132, Valid RMSE: 4.6500
Epoch 25, Train Loss: 4.2905, Valid Loss: 21.8259, Valid RMSE: 4.6667
Epoch 26, Train Loss: 3.9662, Valid Loss: 21.9377, Valid RMSE: 4.6836
Epoch 27, Train Loss: 3.6677, Valid Loss: 22.1226, Valid RMSE: 4.7008
Epoch 28, Train Loss: 3.3915, Valid Loss: 22.3290, Valid RMSE: 4.7181
Epoch 29, Train Loss: 3.1389, Valid Loss: 22.4308, Valid RMSE: 4.7351
Epoch 30, Train Loss: 2.9059, Valid Loss: 22.5236, Valid RMSE: 4.7522
```

This is great! With this, we have the flexibility to tailor the loss function in the training loop as needed. For instance, we can integrate both user and item biases into the model and include regularization terms for these biases (challenging lab exercise).