Matrix factorization for recommender systems

Hyunwoo Seo

2022. 09. 07

Install packages

Install surprise (A python scikit for recommender systems)

```
[1]  from google.colab import drive
     drive.mount('./MyDrive')

Mounted at ./MyDrive
```

```
cd MyDrive/MyDrive/recommender_system

/content/MyDrive/MyDrive/recommender_system
```

```
[3]  !pip install surprise

Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: surprise in /usr/local/lib/python3.7/dist-packages (0.1)
Requirement already satisfied: scikit-surprise in /usr/local/lib/python3.7/dist-packages (from surprise) (1.1.1)
Requirement already satisfied: six>=1.10.0 in /usr/local/lib/python3.7/dist-packages (from scikit-surprise->surprise) (1.15.0)
Requirement already satisfied: joblib>=0.11 in /usr/local/lib/python3.7/dist-packages (from scikit-surprise->surprise) (1.1.0)
Requirement already satisfied: scipy>=1.0.0 in /usr/local/lib/python3.7/dist-packages (from scikit-surprise->surprise) (1.7.3)
Requirement already satisfied: numpy>=1.11.2 in /usr/local/lib/python3.7/dist-packages (from scikit-surprise->surprise) (1.21.6)
```

Load packages

```
[4]  import numpy as np
     import pandas as pd
     import matplotlib.pyplot as plt
     import warnings
     import random

     from IPython.display import display
     from tqdm import tqdm
     from collections import defaultdict

     seed = 0
     np.random.seed(seed)
     random.seed(seed)
     warnings.filterwarnings(action='ignore')
```

- For more information, refer to https://surprise.readthedocs.io/en/stable/.

Netflix prize data (movie recommendation)

Load dataset

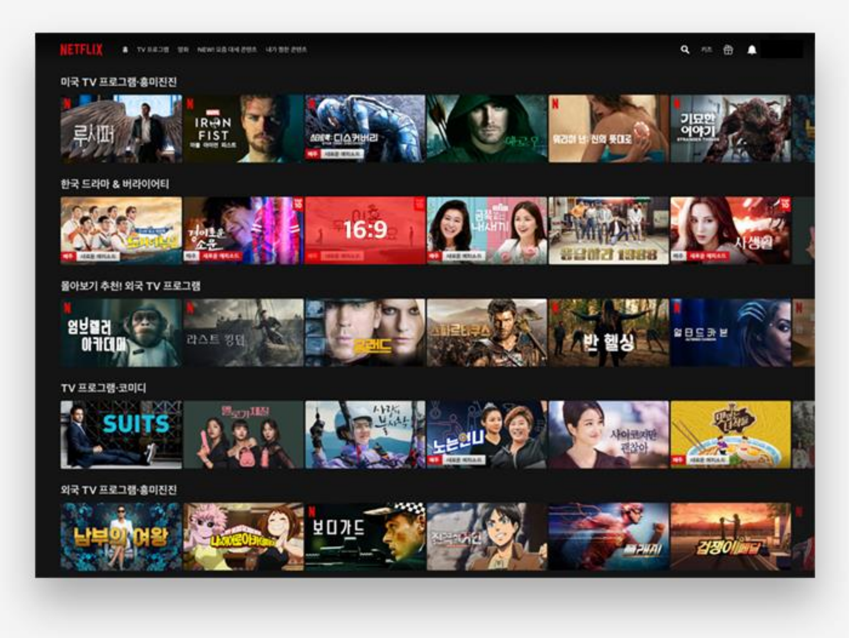
Netflix held the Netflix Prize open competition for the best algorithm to predict user ratings for films.
This is the dataset that was used in that competition. It consists of user id and ratings (1~5) that a user rated to a movie.

```
# Skip date
df = pd.read_csv('combined_data_1.txt', header = None, names = ['Cust_Id', 'Rating'], usecols = [0,1])
df['Rating'] = df['Rating'].astype(float)

print('shape: {}₩n'.format(df.shape))
print('-Dataset examples-')
df.iloc[::5000000, :]
```

```
shape: (24058263, 2)

-Dataset examples-
```

	Cust_Id	Rating
0	1:	NaN
5000000	2560324	4.0
10000000	2271935	2.0
15000000	1921803	2.0
20000000	1933327	3.0

- For more information on Netflix prize data, refer to https://www.kaggle.com/netflix-inc/netflix-prize-data.



Netflix prize data (movie recommendation)

입력 데이터의 형태

각 영화 (item_id)에 대한 리뷰어 (Cust_Id)의 평점이 각 행마다 기록되어 있습니다.

영화 (item_id)를 지칭하는 행은 Rating이 NaN으로 입력되어 있습니다.

```python
display(df.loc[548:554, :])
display(df.loc[694:698, :])
```

	Cust_Id	Rating
548	2:	NaN
549	2059652	4.0
550	1666394	3.0
551	1759415	4.0
552	1959936	5.0
553	998862	4.0
554	2625420	2.0

	Cust_Id	Rating
694	3:	NaN
695	1025579	4.0
696	712664	5.0
697	1331154	4.0
698	2632461	3.0

Add movie ID column

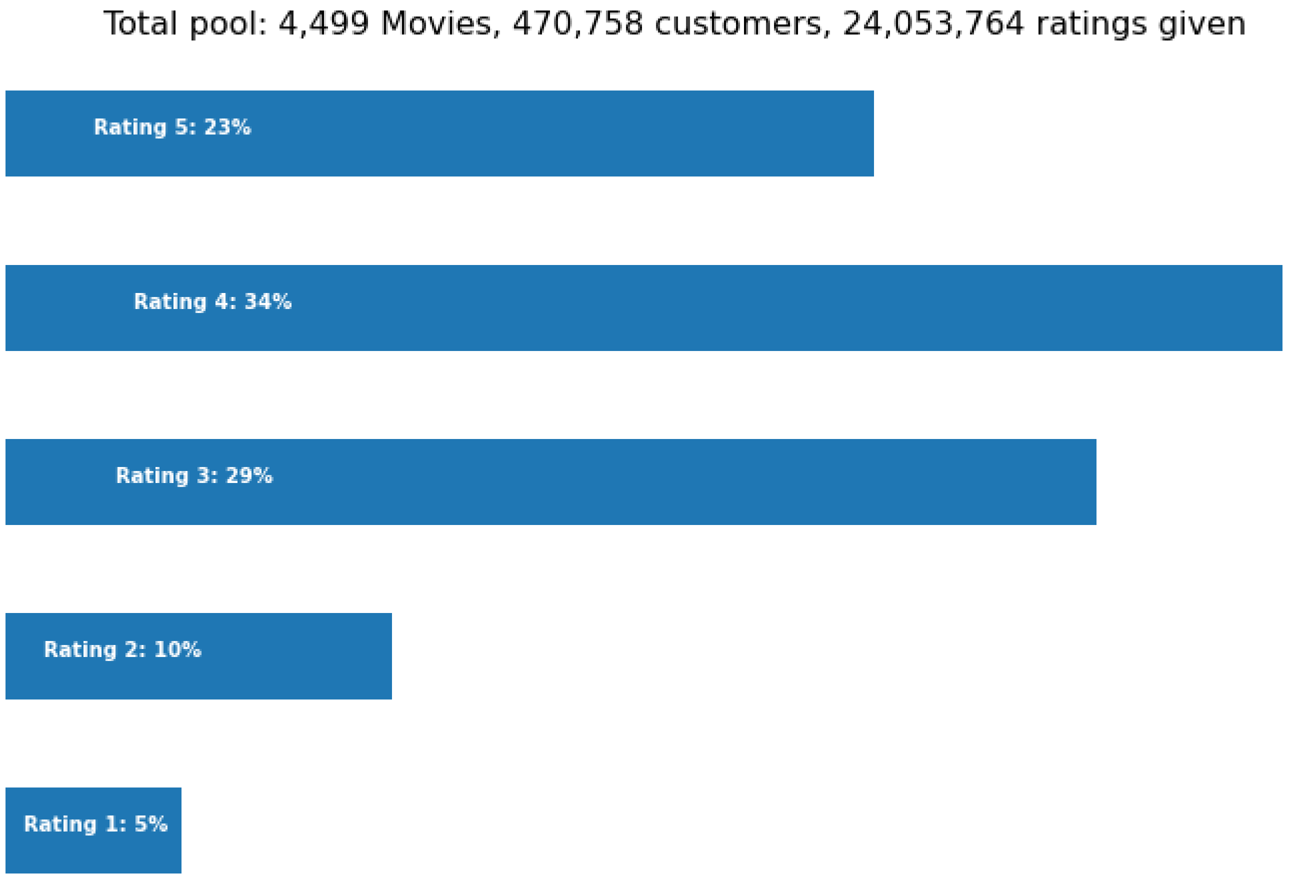
```python
ratings = df.Rating.values
movie_id_arr = np.cumsum(np.isnan(ratings))
df['Movie_Id'] = movie_id_arr
df = df.dropna(axis=0)

print(f'총 rating 수 : {len(df)}')
display(df.loc[::5000,:])
```

총 rating 수 : 24053764

	Cust_Id	Rating	Movie_Id
1	1488844	3.0	1
5006	1189445	1.0	6
10008	2421394	3.0	8
15008	2342811	3.0	8
20008	1834737	3.0	8
...	...	...	...
24034492	819546	2.0	4492
24039492	2570403	1.0	4492
24044493	1042113	4.0	4493
24049496	1059719	3.0	4496
24054496	520014	4.0	4496

4811 rows × 3 columns

Netflix prize data (movie recommendation)



- In the dataset, 4,499 movies (items), 470,758 customers (users), and 24,053,764 ratings (interactions) are given.

Load movie title dataset

Movie ID-Title dataset

```
[8]  df_title = pd.read_csv('movie_titles.csv', encoding = "ISO-8859-1", header = None, names = ['Movie_Id', 'Year', 'Name'])
     df_title.set_index('Movie_Id', inplace = True)
     df_title.head(10)
```

Movie_Id	Year	Name
1	2003.0	Dinosaur Planet
2	2004.0	Isle of Man TT 2004 Review
3	1997.0	Character
4	1994.0	Paula Abdul's Get Up & Dance
5	2004.0	The Rise and Fall of ECW
6	1997.0	Sick
7	1992.0	8 Man
8	2004.0	What the #$*! Do We Know!?
9	1991.0	Class of Nuke 'Em High 2
10	2001.0	Fighter

Build the dataset (user-item matrix)

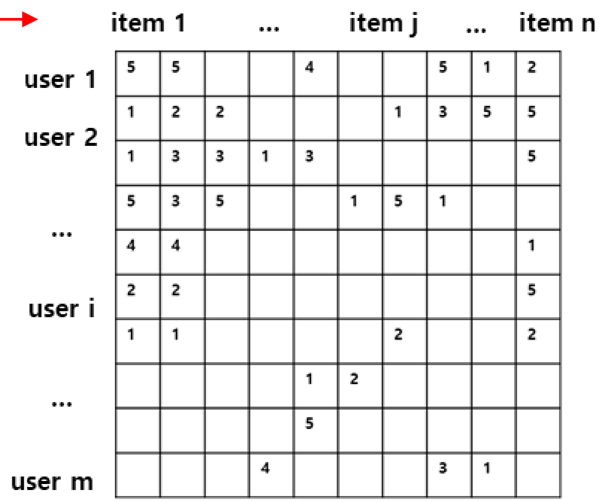
Build the dataset (user-item matrix)

For the fast implementation, I randomly select 100 users and their all interactions.

```
[134]  from surprise import Reader, Dataset
```

```
[135]  reader = Reader()

       # just get interactions of 500 users randomly
       n_users = 500
       all_users = df.Cust_Id.unique()
       sampled_users = np.random.choice(all_users, n_users, replace=False)
       df_sample = df.loc[df['Cust_Id'].isin(sampled_users)]
       print('{} interactions, {} users, {} movies are selected.'.format(len(df_sample), df_sample.Cust_Id.nunique(), df_sample.Movie_Id.nunique()))
       print('only {:.2f}% of possible interactions are observed'.format(len(df_sample) / (df_sample.Cust_Id.nunique()*df_sample.Movie_Id.nunique()) *100))

       data = Dataset.load_from_df(df_sample[['Cust_Id', 'Movie_Id', 'Rating']], reader)
       unobserved_data = data.build_full_trainset().build_anti_testset()
```

```
26626 interactions, 500 users, 2474 movies are selected.
only 2.15% of possible interactions are observed
```



User-item rating matrix

- For the fast implementation, I randomly select 500 users and their all interactions
- 26626 interactions of 500 users to 2474 movies are selected
- User-item matrix is sparse (only 2.15% of possible interactions are given)

Build the dataset (user-item matrix)

Build the dataset (user-item matrix)

For the fast implementation, I randomly select 100 users and their all interactions.

```
[10] from surprise import Reader, Dataset
```

```
reader = Reader()

# just get interactions of 500 users randomly
n_users = 500
all_users = df.Cust_Id.unique()
sampled_users = np.random.choice(all_users, n_users, replace=False)
df_sample = df.loc[df['Cust_Id'].isin(sampled_users)]
print('{} interactions, {} users, {} movies are selected.'.format(len(df_sample), df_sample.Cust_Id.nunique(), df_sample.Movie_Id.nunique()))
print('only {:.2f}% of possible interactions are observed'.format(len(df_sample) / (df_sample.Cust_Id.nunique()*df_sample.Movie_Id.nunique()) *100))

data = Dataset.load_from_df(df_sample[['Cust_Id', 'Movie_Id', 'Rating']], reader)
unobserved_data = data.build_full_trainset().build_anti_testset()
```

```
26124 interactions, 500 users, 2191 movies are selected.
only 2.38% of possible interactions are observed
```

```
[137] # 리뷰어 ID, 영화 ID, 평점, 시간
data.raw_ratings[:10]
```

```
[(1779903, 1, 4.0, None),
 (1765182, 3, 4.0, None),
 (2258411, 3, 3.0, None),
 (47411, 5, 5.0, None),
 (1690808, 5, 4.0, None),
 (229572, 8, 5.0, None),
 (419330, 8, 5.0, None),
 (1367766, 8, 2.0, None),
 (1919323, 8, 2.0, None),
 (469021, 8, 4.0, None)]
```

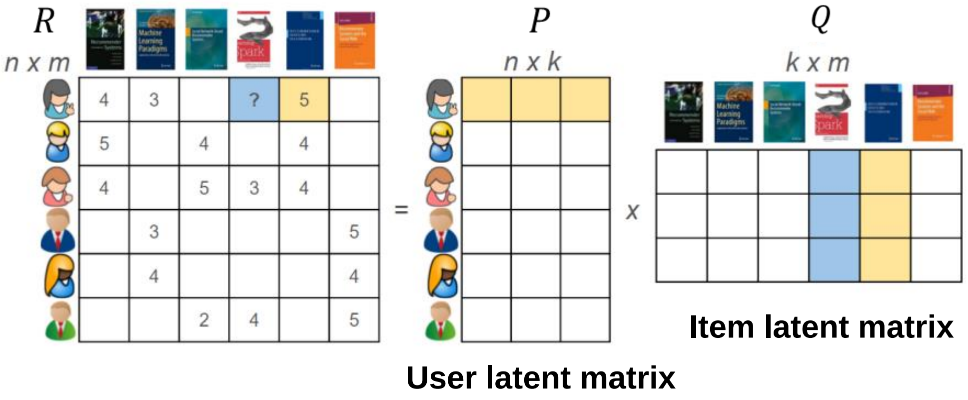
	item 1			...		item j		...		item n
user 1	5	5			4			5	1	2
user 2	1	2	2				1	3	5	5
	1	3	3	1	3					5
	5	3	5			1	5	1		
...	4	4								1
user i	2	2								5
	1	1					2			2
...					1	2				
					5					
user m				4				3	1	

User-item rating matrix

Matrix factorization

- Matrix Factorization for recommender systems

k: the number of latent factors



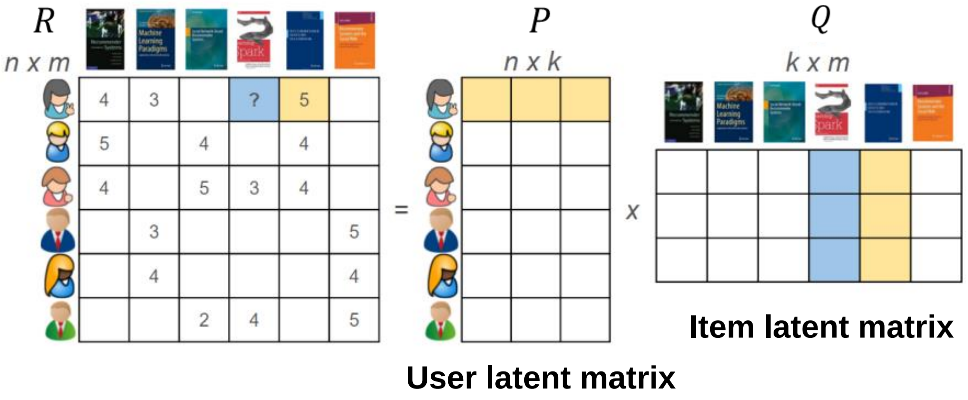
  - $\hat{r}_{ui} = q_i^T p_u$
  - $\hat{r}_{ui} = \mu + b_u + b_i + q_i^T p_u$
  - $minimize \ \sum_{r_{ui} \in R_{train}} (r_{ui} - \hat{r}_{ui})^2 + \lambda(b_i^2 + b_u^2 + ||q_i||^2 + ||p_u||^2)$
    - $where \ b_i: item \ bias, b_u: user \ bias, p_u: latent \ user \ vector, q_i: latent \ item \ vector$

  - Decomposes the rating matrix R (n_users×n_items) into user latent matrix P (n_users×n_latent) and item latent matrix Q (n_items×n_latent).
  - For example, the item latent matrix can represent the genre of the movie.
  - The decomposition facilitates a clear representation of relationships between users and items by mapping from sparse space to dense space.
  - Can introduce biases of users and items.
    - User bias : How much a user rates movies on average.
    - Item bias: How much a movie is rated on average.

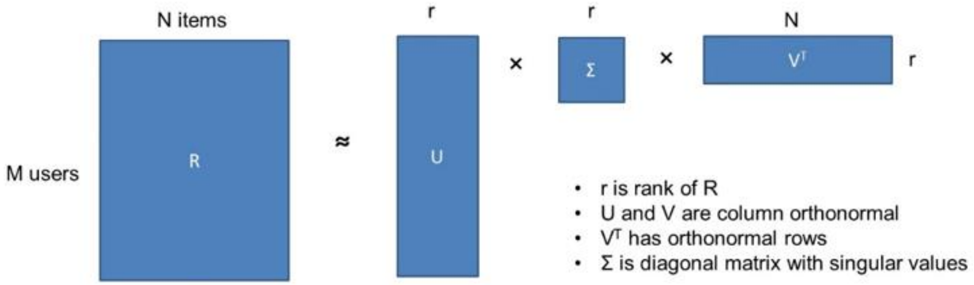
Matrix factorization

- Matrix Factorization for recommender systems
  - $\hat{r}_{ui} = q_i^T p_u$
  - $\hat{r}_{ui} = \mu + b_u + b_i + q_i^T p_u$
  - $minimize \ \sum_{r_{ui} \in R_{train}} (r_{ui} - \hat{r}_{ui})^2 + \lambda(b_i^2 + b_u^2 + ||q_i||^2 + ||p_u||^2)$
    - $where \ b_i: item \ bias, b_u: user \ bias, p_u: latent \ user \ vector, q_i: latent \ item \ vector$

k: the number of latent factors



- SVD (Singular Value Decomposition)



  - U : the relationship b/w users and latent factors
  - Σ : the strength of each latent factor
  - V : the relationship b/w items and latent factors

- NMF (Nonnegative Matrix Factorization)

R (rows: User; columns: Item)

User	W	X	Y	Z
A		4.5	2.0	
B	4.0		3.5	
C		5.0		2.0
D		3.5	4.0	1.0

=

P

A	1.2	0.8
B	1.4	0.9
C	1.5	1.0
D	1.2	0.8

X

Q

W	X	Y	Z
1.5	1.2	1.0	0.8
1.7	0.6	1.1	0.4

  - Note that rating matrix is always positive.
  - All elements in user and item matrices are positive
  - Easy to interpret each latent vector

Model evaluation

```python
from surprise import SVD, NMF, accuracy
from surprise.model_selection import cross_validate

# hyperparameters for training the model
n_factors = 30
n_epochs = 100
biased = True
```

Setting parameters

```python
[12] algo = SVD(n_factors=n_factors, n_epochs=n_epochs, biased=biased, random_state=seed,)
```

```python
[140] cv_result = cross_validate(algo, data, measures=['RMSE', 'MAE'], cv = 5, verbose = True)
```

Cross-validation of SVD

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

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
RMSE (testset)    1.0591  1.0331  1.0488  1.0567  1.0567  1.0509  0.0095
MAE (testset)     0.8204  0.8157  0.8195  0.8281  0.8269  0.8221  0.0047
Fit time          3.08    3.06    3.04    2.98    2.96    3.02    0.05
Test time         0.05    0.03    0.03    0.03    0.03    0.04    0.01
```

```python
[141] algo = NMF(n_factors=n_factors, n_epochs=n_epochs, biased=biased, random_state=seed,)
cv_result = cross_validate(algo, data, measures=['RMSE', 'MAE'], cv = 5, verbose = True)
```

Cross-validation of NMF

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

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
RMSE (testset)    1.8985  1.5922  1.6829  1.4256  1.8489  1.6896  0.1722
MAE (testset)     1.5083  1.2255  1.3277  1.0870  1.4772  1.3251  0.1570
Fit time          4.14    4.12    4.08    4.11    4.05    4.10    0.03
Test time         0.03    0.03    0.04    0.03    0.04    0.04    0.00
```

- Evaluation metrics
  - RMSE (Root Mean Squared Error): $RMSE = \sqrt{\frac{1}{n}\sum_{r_{ui}\in R_{test}}(r_{ui} - \hat{r}_{ui})^2}$
  - MAE (Mean Average Error): $MAE = \frac{1}{n}\sum_{r_{ui}\in R_{test}}|r_{ui} - \hat{r}_{ui}|$
- 5-fold cross-validation results are
  - **SVD:** $\boldsymbol{RMSE = 1.0509, MAE = 0.822}$
  - NMF: $RMSE = 1.6896, MAE = 1.3251$

Find the best hyperparameter by grid search

[152]

```python
from surprise.model_selection import GridSearchCV

param_grid = {'n_factors': [30, 50, 70, 100], 'n_epochs': [50, 100]}
gs = GridSearchCV(SVD, param_grid, measures=['RMSE', 'MAE'], cv = 3)
gs.fit(data)
```

[153]

```python
print('Best RMSE: {:.5f} | params for the best RMSE: {}'.format(gs.best_score['rmse'], gs.best_params['rmse']))
print('Best MAE: {:.5f} | params for the best MAE: {}'.format(gs.best_score['mae'], gs.best_params['mae']))
```

```
Best RMSE: 1.00207 | params for the best RMSE: {'n_factors': 100, 'n_epochs': 50}
Best MAE: 0.78716 | params for the best MAE: {'n_factors': 30, 'n_epochs': 50}
```

See how matrix factorization estimates unknown ratings

- Choose SVD which showed better performance

```python
n_factors, n_epochs = gs.best_params['rmse']['n_factors'], gs.best_params['rmse']['n_epochs']

# 모델 선언
algo = SVD(n_factors=n_factors, n_epochs=n_epochs, biased=biased, random_state=seed,)
# 전체 데이터 훈련 데이터로 설정
trainset = data.build_full_trainset()
# 학습 데이터로 모델 학습
algo.fit(trainset)
# 평점이 없는 (리뷰어, 영화)에 대해 평점 예측
unobserved_pred = algo.test(unobserved_data)
```

```python
user_dict = {i:trainset.to_raw_uid(i) for i in trainset.all_users()} # raw user id: encoded user id
item_dict = {i:trainset.to_raw_iid(i) for i in trainset.all_items()} # raw item id: encoded item id
```

```python
print('(인코딩된 리뷰어 id, 원래 리뷰어 id):', list(user_dict.items())[:5])
print('(인코딩된 영화 id, 원래 영화 id)    :', list(item_dict.items())[:5])
```

```
(인코딩된 리뷰어 id, 원래 리뷰어 id): [(0, 1779903), (1, 1765182), (2, 2258411), (3, 47411), (4, 1690808)]
(인코딩된 영화 id, 원래 영화 id)    : [(0, 1), (1, 3), (2, 5), (3, 8), (4, 9)]
```

- user_dict, item_dict: 실제 사용자/영화 ID와 데이터셋 내에서 인코딩된 사용자/영화 ID 딕셔너리

See how matrix factorization estimates unknown ratings

- Select a user and an item as a sample

pu: user latent factor, qi: item latent factor

bu: user bias, bi: item bias

```
uid = 0
iid = 0

pu = algo.pu[uid]
qi = algo.qi[iid]
bu = algo.bu[uid]
bi = algo.bi[iid]

print('사용자 잠재 벡터 (p_u) 형태=(총 사용자 수, 잠재 차원)={}: ₩nEx) User factor of user {} (실제 id: {})₩n{}₩n'.format(algo.pu.shape, uid, user_
print('아이템 잠재 벡터 (q_i) 형태=(총 아이템 수, 잠재 차원)={}: ₩nEx) Item factor of movie {} (실제 id: {})₩n{}₩n'.format(algo.qi.shape, iid, item
print('사용자 편향 (b_u) 형태=(총 사용자 수, )={}: ₩nEx) User bias of user {} (실제 id: {}): {}₩n'.format(algo.bu.shape, uid, user_dict[uid], bu))
print('아이템 편향 (b_i) 형태=(총 아이템 수, )={}: ₩nEx) Item bias of movie {} (실제 id: {}): {}'.format(algo.bi.shape, iid, item_dict[iid], bi))
```

```
사용자 잠재 벡터 (p_u) 형태=(총 사용자 수, 잠재 차원)=(500, 100):
Ex) User factor of user 0 (실제 id: 1779903)
[ 2.15084979e-01 -1.58366360e-01  5.70734887e-01  4.35013153e-01
  4.51851090e-01 -8.36742012e-02  2.94620783e-02  1.69268771e-01
 -3.72217748e-01 -2.29711141e-01  5.18793000e-02  3.72975839e-01
 -6.86515316e-02  4.77949629e-02  2.87759058e-01 -3.04398818e-01
  7.72602489e-01 -9.02894027e-02  8.99296111e-02  1.35024966e-01
 -3.03112840e-01  1.38993185e-01 -1.27194196e-01 -2.19597031e-01
  4.42953476e-01 -2.95362617e-01  1.61290334e-01  1.21088798e-01
  6.95178378e-02  2.94389066e-01 -1.44552344e-01  2.15736058e-01
 -1.32971586e-01 -2.56606237e-01 -1.80961515e-01  2.63552705e-01
  1.25598283e-01  3.70664227e-01 -2.34301048e-02  1.64310884e-01
 -4.73021211e-01 -6.39759316e-01 -3.61551028e-01  6.53412600e-01
 -2.53493423e-01 -1.15362582e-01 -6.26913882e-04 -1.57178054e-01
 -1.80518757e-01 -4.59978909e-02 -3.49804086e-01 -2.88109626e-02
 -1.34376786e-01 -4.86491207e-01 -1.21397491e-01  1.59348491e-01
 -1.53722438e-01  2.47518011e-02 -9.99466947e-02 -2.62601400e-01
 -1.57647401e-01 -6.00506011e-02 -1.93123148e-01 -5.49997189e-02
  4.73888873e-01 -1.37854955e-01 -2.79349128e-01  1.95911518e-01
 -7.41346456e-02 -3.11659691e-01  3.25899694e-01 -1.78137410e-01
  1.08916741e-01 -3.87465501e-01  3.24270719e-01 -4.16456924e-01
 -9.80356583e-02  5.80300092e-02  1.94987828e-01 -2.65825355e-01
 -2.92755462e-01  3.50847419e-01  1.51659499e-01 -2.22017132e-01
  5.32494493e-01  2.86572084e-01  4.32845463e-01  1.12038443e-01
 -2.55195364e-01  2.96173825e-02  3.48929671e-02  3.13397755e-01
 -2.15604200e-01  1.48646301e-01  2.45378419e-01  5.28303953e-01
 -2.37967157e-01  3.01802960e-01 -2.27555218e-01  3.38814641e-01]
```

```
아이템 잠재 벡터 (q_i) 형태=(총 아이템 수, 잠재 차원)=(2474, 100):
Ex) Item factor of movie 0 (실제 id: 1)
[ 0.03118569  0.15823942 -0.11745345 -0.01398305 -0.1686289  -0.09676564
  0.04131564 -0.04747524 -0.22318781 -0.18941386  0.04196442 -0.00896561
  0.20457323  0.20486869  0.04845433 -0.16333017  0.09713182 -0.0934273
 -0.037339    0.21171056 -0.07503193 -0.03005105 -0.04589698  0.08458869
 -0.03157934  0.0214926  -0.08212937 -0.13618437 -0.06072453  0.21388015
 -0.18703256 -0.06712876 -0.05084275 -0.05150505 -0.09513854 -0.08618602
  0.05034062 -0.04039532  0.10461177  0.10470294 -0.02538849 -0.0195008
 -0.11888891  0.17194575 -0.00961996 -0.03006054  0.05743363 -0.02975336
  0.09134327 -0.01275188 -0.036991   -0.04159174 -0.11066434  0.04522668
 -0.0189111  -0.17757561  0.12582342 -0.02429241 -0.02184705 -0.03860453
 -0.04576679  0.07827723  0.12499379 -0.01770302  0.03283828 -0.14510405
 -0.05524539  0.1536903  -0.00441691  0.02492228 -0.14359572  0.18055776
  0.01374827 -0.18498622  0.16285581  0.11688934 -0.13004971 -0.08149852
  0.00205881 -0.01545119  0.03335275 -0.02086776 -0.07418131 -0.26156913
 -0.03877804 -0.09494778  0.21462496 -0.00892484  0.0160932   0.16217652
  0.13269777 -0.02997385 -0.15997215 -0.14533302 -0.08939973  0.05433299
 -0.10459304 -0.09869952 -0.03246414 -0.02023327]

사용자 편향 (b_u) 형태=(총 사용자 수, )=(500,):
Ex) User bias of user 0 (실제 id: 1779903): -0.37577986716460926

아이템 편향 (b_i) 형태=(총 아이템 수, )=(2474,):
Ex) Item bias of movie 0 (실제 id: 1): 0.1341331673099895
```

See how matrix factorization estimates unknown ratings

- Prediction by SVD.predict method in Surprise

```
[159] # prediction by SVD.predict method
      algo.predict(user_dict[uid], item_dict[iid])
```

```
Prediction(uid=1779903, iid=1, r_ui=None, est=3.862934630847635, details={'was_impossible': False})
```

- Manual calculation
  - $\hat{r}_{ui} = \mu + b_u + b_i + q_i^T p_u$

```
[160] # manual calculation
      trainset.global_mean + bu + bi + np.dot(pu, qi)
```

```
3.862934630847635
```

Top-n recommendation

Top-n recommendation

- Recommend n items that a user is expected to like (items with n highest predicted ratings) to the user who has ever rated.

```python
def get_top_n(predictions, user_dict, user, n=10):
    """
    Return the top-N recommendation for each user from a set of predictions.
    [(raw item id, rating estimation), ...] of size n.
    """
    # First map the predictions to each user.
    top_n = []
    for uid, iid, true_r, est, _ in predictions:
        if user_dict[user]==uid:
            top_n.append((iid, est))
    top_n.sort(key=lambda x: x[1], reverse=True)
    top_n = top_n[:n]

    print('Top-{} recommendations for user {} (실제 id: {})'.format(n, user, user_dict[user]))
    for iid, est in top_n:
        print('r_est: {:.5f} | movie ID:{:>5} | name: {}'.format(est, iid, df_title.loc[iid].Name))

    return top_n
```

Top-n recommendation

Top-n recommendation

- Recommend n items that a user is expected to like (items with n highest predicted ratings) to the user who has ever rated.

```
[162]  top_n = get_top_n(unobserved_pred, user_dict, user=0, n=10)
```

```
Top-10 recommendations for user 0 (실제 id: 1779903)
r_est: 4.69695 | movie ID: 1495 | name: Alias: Season 1
r_est: 4.58172 | movie ID: 1295 | name: Strange Brew
r_est: 4.57059 | movie ID: 2848 | name: The Hustler
r_est: 4.55000 | movie ID: 3521 | name: Coupling: Season 1
r_est: 4.45329 | movie ID: 1110 | name: Secondhand Lions
r_est: 4.44316 | movie ID: 3456 | name: Lost: Season 1
r_est: 4.43320 | movie ID: 2319 | name: The Looney Tunes Golden Collection: Vol. 1
r_est: 4.39042 | movie ID:  833 | name: White Squall
r_est: 4.37634 | movie ID:  224 | name: Midsomer Murders: Blood Will Out
r_est: 4.37021 | movie ID: 2057 | name: Buffy the Vampire Slayer: Season 6
```

```
[163]  top_n = get_top_n(unobserved_pred, user_dict, user=40, n=10)
```

```
Top-10 recommendations for user 40 (실제 id: 2361784)
r_est: 5.00000 | movie ID:  127 | name: Fatal Beauty
r_est: 5.00000 | movie ID:  241 | name: North by Northwest
r_est: 5.00000 | movie ID:  270 | name: Sex and the City: Season 4
r_est: 5.00000 | movie ID:  468 | name: The Matrix: Revolutions
r_est: 5.00000 | movie ID: 1395 | name: Charade
r_est: 5.00000 | movie ID: 1495 | name: Alias: Season 1
r_est: 5.00000 | movie ID: 1499 | name: FLCL
r_est: 5.00000 | movie ID: 1625 | name: Aliens: Collector's Edition
r_est: 5.00000 | movie ID: 1642 | name: Casino: 10th Anniversary Edition
r_est: 5.00000 | movie ID: 1798 | name: Lethal Weapon
```

Movies with similar preference

- Movies with similar preference
  - This function gives a list of n movies of similar preference of users given a query movie.

```
[111]
def similar_movies(algo, item_dict, item, n):

    query_repr = algo.qi[item] # 입력한 영화의 잠재벡터
    key_repr = algo.qi # 모든 영화의 잠재벡터

    sim = np.dot(query_repr, key_repr.T)/(np.sqrt(sum(query_repr**2))*np.sqrt(np.sum(key_repr**2, axis=1))) # 입력한 영화와의 코사인 유사도

    # 입력한 영화와 사용자 선호에 대한 유사도가 가장 높은 n개 영화 id
    most_similar_movies = np.argsort(sim)[::-1][:n+1]
    query_idx = np.argwhere(most_similar_movies==item)
    most_similar_movies = np.delete(most_similar_movies, query_idx)
    most_similar_movies = [item_dict[i] for i in most_similar_movies]

    print(f'Query movie : {df_title.loc[item_dict[item]].Name}₩n')
    return df_title.loc[most_similar_movies]
```

Movies with similar preference

- Movies with similar preference
  - This function gives a list of n movies of similar preference of users given a query movie.

```
[165] 1 similar_movies(algo, item_dict, 10, 10)
```

Query movie : By Dawn's Early Light

Movie_Id	Year	Name
2286	1980.0	Fade to Black
1558	1990.0	Rocky V
1424	2000.0	Beautiful Joe
1914	2002.0	Damaged Care
3479	2002.0	Deceived
3309	2001.0	The Pledge
3557	2000.0	Gun Shy
2037	1985.0	Fright Night
2771	1984.0	Top Secret!
1318	1985.0	Baby: Secret of the Lost Legend

```
[166] 1 similar_movies(algo, item_dict, 102, 10)
```

Query movie : Ruby's Bucket of Blood

Movie_Id	Year	Name
4144	2001.0	The Brothers
1816	2003.0	Northfork
2828	1999.0	Judgment Day
3405	2002.0	The Dead Zone: Season 2
809	2002.0	Left Behind II: Tribulation Force
587	1983.0	The Scarlet and the Black
2657	1963.0	Heavens Above!
606	1968.0	Quatermass and the Pit
4010	1963.0	The Leopard (Original Italian Version)
3733	1963.0	Spencer's Mountain

Visualizing the movie Factors Using t-SNE

Visualizing the movie Factors Using t-SNE

[117]

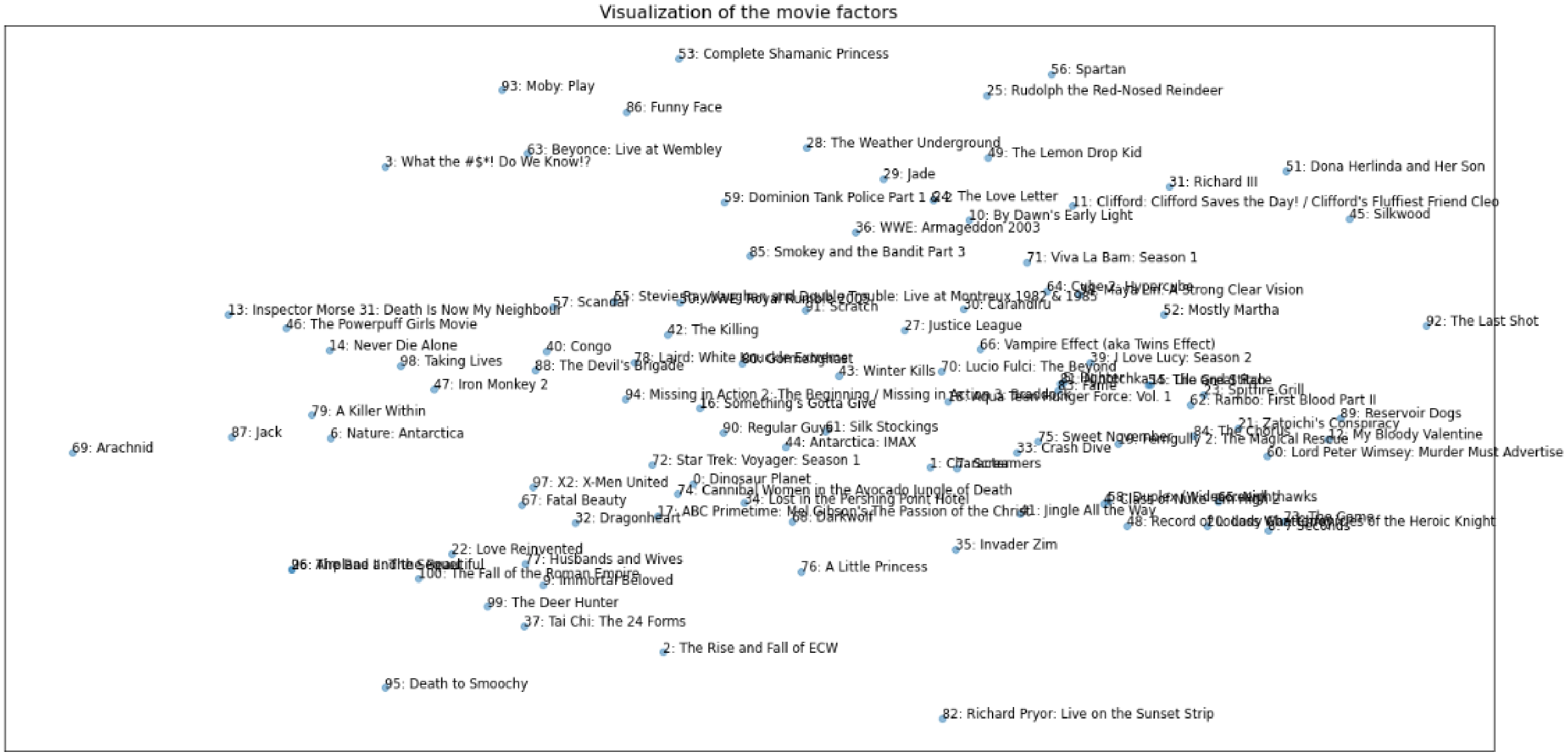
```python
from sklearn.manifold import TSNE

tsne = TSNE(n_components=2, perplexity=10, n_iter=1000, verbose=3, random_state=seed)
movie_embedding = tsne.fit_transform(algo.qi)
```

```
[t-SNE] Computing 31 nearest neighbors...
[t-SNE] Indexed 2191 samples in 0.001s...
[t-SNE] Computed neighbors for 2191 samples in 0.156s...
[t-SNE] Computed conditional probabilities for sample 1000 / 2191
[t-SNE] Computed conditional probabilities for sample 2000 / 2191
[t-SNE] Computed conditional probabilities for sample 2191 / 2191
[t-SNE] Mean sigma: 0.214650
[t-SNE] Computed conditional probabilities in 0.037s
[t-SNE] Iteration 50: error = 93.9240341, gradient norm = 0.2331061 (50 iterations in 1.104s)
[t-SNE] Iteration 100: error = 97.1831055, gradient norm = 0.1953556 (50 iterations in 1.792s)
[t-SNE] Iteration 150: error = 97.5197449, gradient norm = 0.1974167 (50 iterations in 2.333s)
[t-SNE] Iteration 200: error = 97.9256134, gradient norm = 0.1862506 (50 iterations in 2.314s)
[t-SNE] Iteration 250: error = 98.0254593, gradient norm = 0.1888251 (50 iterations in 1.747s)
[t-SNE] KL divergence after 250 iterations with early exaggeration: 98.025459
[t-SNE] Iteration 300: error = 3.9076753, gradient norm = 0.0024458 (50 iterations in 1.734s)
[t-SNE] Iteration 350: error = 3.6638875, gradient norm = 0.0007761 (50 iterations in 2.262s)
[t-SNE] Iteration 400: error = 3.5573983, gradient norm = 0.0004652 (50 iterations in 1.367s)
[t-SNE] Iteration 450: error = 3.4994621, gradient norm = 0.0002748 (50 iterations in 0.874s)
[t-SNE] Iteration 500: error = 3.4644165, gradient norm = 0.0002687 (50 iterations in 0.945s)
[t-SNE] Iteration 550: error = 3.4428134, gradient norm = 0.0001704 (50 iterations in 0.798s)
[t-SNE] Iteration 600: error = 3.4291749, gradient norm = 0.0001372 (50 iterations in 0.866s)
[t-SNE] Iteration 650: error = 3.4196467, gradient norm = 0.0001406 (50 iterations in 0.922s)
[t-SNE] Iteration 700: error = 3.4133325, gradient norm = 0.0001069 (50 iterations in 1.602s)
[t-SNE] Iteration 750: error = 3.4088683, gradient norm = 0.0000933 (50 iterations in 0.926s)
[t-SNE] Iteration 800: error = 3.4053757, gradient norm = 0.0000955 (50 iterations in 0.811s)
[t-SNE] Iteration 850: error = 3.4021192, gradient norm = 0.0000916 (50 iterations in 1.734s)
[t-SNE] Iteration 900: error = 3.3984160, gradient norm = 0.0000918 (50 iterations in 0.791s)
[t-SNE] Iteration 950: error = 3.3955331, gradient norm = 0.0001001 (50 iterations in 0.788s)
[t-SNE] Iteration 1000: error = 3.3924026, gradient norm = 0.0000773 (50 iterations in 0.900s)
[t-SNE] KL divergence after 1000 iterations: 3.392403
```

Visualizing the movie Factors Using t-SNE

[171]

```python
n_samples = 100
projection_plot = projection.loc[:n_samples]
fig, ax = plt.subplots(figsize=(24, 12))
ax.scatter(projection_plot.x.values, projection_plot.y.values, alpha=0.5)
ax.set_xticks([])
ax.set_yticks([])
ax.grid(False)
ax.set_title('Visualization of the movie factors', size=16)
for i, (mid, txt) in enumerate(zip(projection_plot.Movie_Id_Encoded, projection_plot.Title)):
    ax.annotate('{}: {}'.format(mid, txt), (projection_plot.x.values[i], projection_plot.y.values[i]), fontsize=11.5)
```



Matrix factorization for implicit feedbacks

- Challenges of matrix factorization
  - Sparsity : a few observed ratings compared to unobserved ratings.
  - Cold-start problem: unable to recommend items to a new user, unable to recommend a new items to users.

- Matrix factorization for implicit feedbacks
  - Users do not express their explicit preference in the online environment. Models should infer users' preference from implicit feedbacks (ex. watch history, shopping list).
  - Deep learning-based recommender systems infer users' preference well from implicit feedbacks, by using meta information of users and items.

감사합니다