project 4 report

2017030464 한현진

OS : Window 10 **언어버전** : Python 3.10.1

IDE : Pycharm Community Edition 2021.2.2

ü Summary of your algorithm

SGD 방법을 사용한 matrix factorization 방식으로 collaborative filtering을 해서 base 데이터를 바탕으로 test 데이터에 존재하는 user – item 짝의 rating 을 예측했습니다. 자세한 설명은 아래 코드 설명과 같이 진행하겠습니다.

ü Detailed description of your codes (for each function)]

코드의 흐름으로 설명 드리겠습니다.

main 함수

```
def main():
    # 입력 받아오는 부분
    args = sys.argv[1:]
    base_name = args[0]
    test_name = args[1]

    base_data = pd.read_csv("./data/" + base_name, delimiter='\t', names=['user_id', 'item_id', 'rating', 'time_stamp'])
    test_data = pd.read_csv("./data/" + test_name, delimiter='\t', names=['user_id', 'item_id', 'rating', 'time_stamp'])
    del base_data['time_stamp']
    del test_data['time_stamp']
    test_data['rating'] = 0

    # merge 하는 이유: test 데이터에는 존재하고 base에 없는 데이터가 존재할 수도 있기 때문이다.
    data = pd.merge(base_data, test_data, how='outer', on=['user_id', 'item_id', 'rating'])
    rating_table = data.pivot_table('rating', index='user_id', columns='item_id').replace(0, np.nan)

    print("cf_start")
    cf_pred_matrix = SGD_matrix_factorization(rating_table)
    cf_result = pd.DataFrame(cf_pred_matrix, columns=rating_table.columns, index=rating_table.index)
    print("cf_end")
```

우선 base 파일과, test 파일을 불러와서, column 의 이름이 'user_id', 'item_id', 'rating', 'time_stamp' 인 data frame 으로 만들어 줍니다.

그리고 time_stamp 값들은 저의 코드 같은 경우에 필요없어보여서, 그냥 삭제해줬습니다.

Test_data 의 rating 값은 사용하지 말라고 하셔서 우선 0 으로 초기화 했습니다.

그리고 base_data 와 test_data 를 merge 해줘서 data 라는 변수로 지정해줍니다.
merge 를 하는 이유는, base_data 에 없는 user 나 item 이 test_data 에는 존재할 수 있어서, matrix factorization 을 하기 위한 matrix 를 생성할 때, 누락된 데이터가 없게 하기 위함입니다.

Merge 를 다 끝냈다면, 해당 데이터프레임을 기반으로, user_id 가 index 이고, item_id 가 column 인 rating_table 을 만들어줍니다. 이 때, test_data 에 있던 rating 들은 처음에 0 으로 초기화시켜줬는데, 0 도 값으로 인식될 수 있기 때문에, np.nan 으로 replace 를 해줬습니다.

제가 이 과제를 하면서 강의에서 배웠던 좀 더 예측을 잘하게 해주는 여러 방법을 시도한 것이 있습니다. 그 중 하나가, 아래 이미지에 보이는 코드인데, 이는 삭제한 코드이긴 합니다만 이러한 시도를 했었다는 것은 알아주셨으면 좋겠습니다.

```
# 실험 - base엔 없고 test에만 있는 자료들 각 평균값으로 넣어주는 방법
base_user = base_data['user_id']
base_item = base_data['item_id']
base_user = set(base_user)
base_item = set(base_item)
test_user = test_data['user_id']
test_item = test_data['item_id']
test_user = set(test_user)
test_item = set(test_item)
not_in_base_user = test_user - base_user
not_in_base_item = test_item - base_item
user_mean_values = np.nanmean(rating_table.values, axis=0)
item_mean_values = np.nanmean(rating_table.values, axis=1)
for user in not_in_base_user:
    rating_table.loc[user, :] = user_mean_values
for item in not_in_base_item:
    rating_table[item] = item_mean_values
```

test_data 에만 존재하는 데이터는 결국 matrix 를 형성할 때, 그것이 user 던, item 이던 rating 기록이 없는 것이기 때문에, cold start 가 될 수 있다는 점을 고려해서, 먼저 이 test_data 에만 존재하는 데이터들에는 user 가 test 에만 있는 user 였다면, 모든 각각의 item 의 평균을 구해서 해당 user 에 대한 각각의 item 의 rating 으로 넣어주고, item 이 test 에만 있는 item 이였다면, 모든 각각의 user 의 평균을 구해서 해당 item 에 대한 각각의 user 의 rating 으로 넣어주고, matrix factorization 을 진행하는 것이였습니다. Content based 방식을 사용하면 더 좋았겠지만, 맞는 데이터를 찾을 수 없어서 시도해보지는 못했습니다.

하지만 막상 이 방법을 사용해서 MF 를 진행한 것이 이 방법을 사용하지 않고 MF 를 진행한 결과와 PA4.exe 채점파일에서 오히려 rmse 가 근소하지만 약 0.01 정도 더 올라서 더 안좋은 방향이 되어서 그냥 삭제해줬습니다.

그래서 그냥 base_data 와 test_data 를 합치기만 했을 뿐인 data 로 SGD_matrix_factorization 을 해서 MF 를 진행해줬습니다.

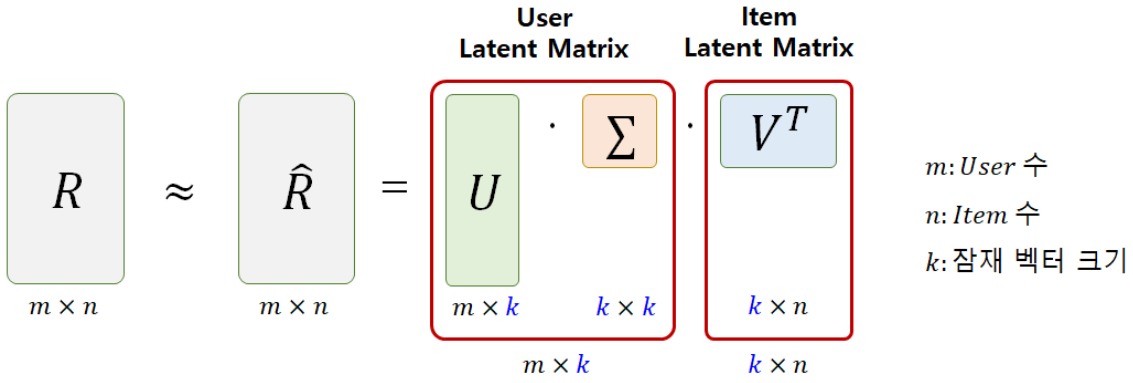
SGD_matrix_factorization 함수

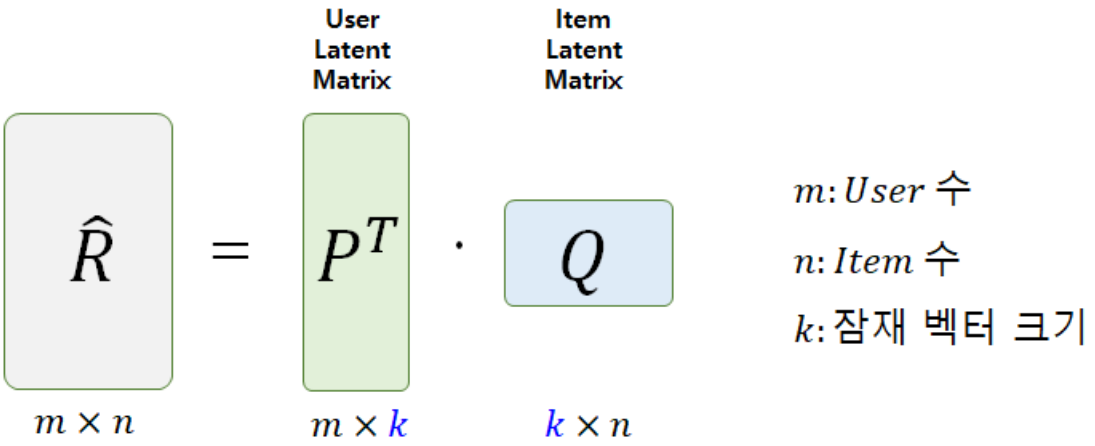
SGD Matrix factorization 을 구현함에 있어서,

https://sungkee-book.tistory.com/12

https://www.blossominkyung.com/archives/matrix-factorization

이 두 블로그를 참고했습니다.





*P와 Q는 정규분포 기반의 랜덤 값들로 초기화

```
def SGD_matrix_factorization(table):
    R = table.values
    num_users, num_items = R.shape
    K = 3
    np.random.seed(2017030464)

    U = np.random.normal(0, 1/K, (num_users, K))
    I = np.random.normal(0, 1/K, (num_items, K))

    not_nans = [(i, j, R[i, j]) for i in range(num_users) for j in range(num_items) if not pd.isna(R[i, j])]
```

위에 보이는 잠재벡터를 만드는함수입니다.

R은 함수의 매개변수로 불러온 table을 numpy형식으로 변형한 것이고,

U는 user latent matrix, I는 item latent matrix 입니다.

K값은 위에 이미지에 써있듯이, 잠재 벡터의 크기인데, 이것을 3으로도 해보고, 10, 50, 100, 1000으로도 해봤으나, 값들의 차이는 미미했지만, 값이 커질수록 성능(MF하는데 걸리는 시간)이 오래 걸리게 되고, 어떤 것은 PA4의 채점 rmse의 결과가 더 나빠지기도 해서 그냥 강의 자료에 있는 3정도로 잡아뒀습니다.

그리고 이미지에 적혀있는대로 U,I matrix 에는 랜덤값이 들어가는데 랜덤시드도 그냥 1이든 0이든 별반 다를 게 없어서, 그냥 제 학번으로 썼습니다.

후에, nan 값이 아닌 즉, base data에 존재한 실제 rating 값들과 그 위치정보를 저장한 not_nans를 만들었습니다.

2. Adding Biases

사용자나 아이템별로 평점에 편향이 있을 수 있다. 예를 들어, 사용자 A는 점수를 후하게 주고 반면에 사용자 B는 점수를 짜게 준다던지, 아니면 어떤 영화는 유명한 명작이라 사용자의 취향과 별개로 점수가 높고, 어떤 영화는 그렇지 않을 수 있다. [그림7]을 보면, Adding Bias에서 예측 평점은 학습 데이터 전체 평점의 평균(μ)에 사용자가 가진 편향(b_u), 아이템이 가진 편향(b_i), 그리고 두 잠재벡터의 내적들로 구성되어 있다. 모든 평점의 평균을 더해주는 이유는 대부분의 아이템이 평균적으로 μ 정도의 값은 받는다는 것을 감안해주기 위함이다.

$$\hat{r}_{u,i} = \mu + b_u + b_i + p_u^T q_i$$

```
# 편향 추가
b_U = np.zeros(num_users)
b_I = np.zeros(num_items)
mean = np.nanmean(R)
```

블로그에 이렇게 편향을 고려하는 방법도 적혀있어서 bias도 추가해줬습니다. 이 또한 PA4.exe의 결과가 더 나빠졌다면 그냥 삭제해버렸겠지만, 0.004 ~ 0.02정도 rmse가 낮아져서 성능이 더 좋아졌기 때문에, 그냥 뒀습니다. 우선 초기값은 편향을 모르니, user와 item수에 맞게 편향값을 0으로 만들어줍니다. Mean은 nan을 제외한 R의 전체평균입니다.

앞서 편미분을 통해 구한 Gradient를 현재 상태의 파라미터 값에서 빼줌으로써 업데이트를 진행한다([그림. 6] 참고). 여기서 에타(η)는 Learning Rate를 의미한다. Learning Rate란, 파라미터를 업데이트 할 때 얼마나 크게 변화시킬지를 정하는 하이퍼 파라미터이다. <더보기>의 Gradient 그림을 보면 weight라는 동그라미가 아래로 이동하는데, 그 이동하는 '보폭'을 얼마나 크게 할 것인가를 정한다고 생각하면 된다.

2) 정규화

두 번째는 과적합을 방지하는 정규화 텀이다. **모델이 학습함에 따라 파라미터의 값(weight)이 점점 커지는데, 너무 큰 경우에는 학습 데이터에 과적합하게 된다.** 이를 방지하기 위하여, 학습 파라미터인 p와 q의 값이 너무 커지지 않도록 규제를 하는 것이 정규화이다. MF에서는 L2 정규화를 사용하였다. 파라미터 값이 커지면 p벡터의 제곱과 q벡터의 제곱 합도 커질 것이고 이는 목적 함수도 커지게 하므로 패널티를 주는 것이다. 람다는 목적 함수에 정규화 텀의 영향력을 어느 정도로 줄 것인지 정하는 하이퍼 파라미터이다. 람다가 너무 작으면 정규화 효과가 적고, 너무 크면 파라미터가 제대로 학습되지 않아서 언더피팅(Underfitting)이 발생한다.

```
learning_rate = 0.01
lambda_ = 0.01
```

Learning_rate와 lambda_의 값을 초기화 해줍니다. 위에 설명이 되어있습니다. 하지만 수치는 어떻게 줘야할지 감이 잘 안잡혀서 인터넷을 찾아본 결과 보통 0.01 값을 줬습니다. 실험정신으로 1~0.001까지 값을 넣어봤는데, 1~0.1 까지는 rmse가 1을 넘어서까지 커져서 안되겠고, 0.01부터 0.001까지는 비슷했습니다. 근데 사람들이 주로 사용하는 값엔 다 이유가 있겠거니 하고 이렇게 설정해줬습니다.

$$b_u \leftarrow b_u + \eta \cdot (e_{u,i} - \lambda b_u)$$

$$b_i \leftarrow b_i + \eta \cdot (e_{u,i} - \lambda b_i)$$

$$p_u \leftarrow p_u + \eta \cdot (e_{u,i} q_i - \lambda p_u)$$

$$q_i \leftarrow q_i + \eta \cdot (e_{u,i} p_u - \lambda q_i)$$

```
count = 0
iter = 0
before_rmse = 0
while count < 3 and iter < 50:
    iter += 1
    for user, item, rating in not_nans:
        # 실제 값과 예측 값의 차이인 오류 값 구함
        pred_R = mean + b_U[user] + b_I[item] + np.dot(U[user, :], I[item, :].T)
        error = rating - pred_R

        b_U[user] += learning_rate * (error - lambda_ * b_U[user])
        b_I[item] += learning_rate * (error - lambda_ * b_I[item])
        U[user, :] += learning_rate * (error * I[item, :] - lambda_ * U[user, :])
        I[item, :] += learning_rate * (error * U[user, :] - lambda_ * I[item, :])

    rmse = check_rmse(R, U, I, not_nans, b_U, b_I, mean)
    print(str(iter) + " times MF's rmse: " + str(rmse))
    if before_rmse != 0 and before_rmse - rmse < 0.001:
        count += 1
    before_rmse = rmse
print("MF was performed " + str(iter) + " times.")
pred_matrix = mean + b_U[:, np.newaxis] + b_I[np.newaxis, :] + np.dot(U, I.T)

return pred_matrix
```

$$e_{ui} = r_{ui} - q_i^T p_u$$

$$\hat{r}_{u,i} = \mu + b_u + b_i + p_u^T q_i$$

(편향을 적용했을 땐, 뒤에 간접행렬을 곱하는 부분이 오른쪽 이미지에 적힌 식으로 대체됩니다.)

MF를 수행하는 반복문 입니다.
count는 이전 rmse와 0.001 이하로 차이난 경우의 횟수입니다.
iter는 MF를 반복한 횟수입니다.
before_rmse는 직전 예측matrix의 rmse입니다.

Pred_R, error, b_U, b_I, U, I 모두 블로그를 참고하여 수식대로 코드를 작성했습니다.
그리고 check_rmse 함수로 자체 rmse를 체크하여 base 데이터에 적혀있던 rating들과 예측

matrix간에는 얼마나 rmse가 측정되는지 확인을 해봤습니다. 이를 통해 반복문을 설정한 것입니다.

처음 코드를 작성할 땐, 한번만이라도, rmse가 전이랑 0.001 미만으로 차이난다면 바로 MF를 중단하는 것이였는데, zero injection을 실험해보면서 초반에 바로 0.001 정도만 차이나지만 그 이후에 점점 더 차이를 좁혀가는 경우도 있음을 확인하고 3번정도 기회를 주었습니다.

그럼 무조건 MF를 많이하면 좋은건가... 한다면 그것또한 아닙니다. 제 코드를 바탕으로 주어진 데이터를 학습시켜보면 MF를 약 30 후반대에서 40 초중반 횟수까지 반복을 하는데, 코드를 임의로 수정해서 100번도 해보았지만 40번 MF 할 때랑 PA4 챗점상의 rmse 결과 별로 차이나지 않았고 어떤 것은 많이했는데 오히려 rmse가 증가했기도 했고, 또 어떤때는 점점 자체 rmse가 낮아지다가도 갑자기 증가하면서 업데이트가 되는 경우도 생겨 성능이 더 나빠졌는 경우도 있었고, piazza 질문목록을 보니, 성능도 3분을 넘어가는 건 좀 그렇다고 하셨기 때문에 만약 50번 MF할 때 까지도 matrix가 좁혀지지 않았다면, 그냥 반복을 끝내도록 limit를 걸어줬습니다.

중간에 print 문은 이 프로그램이 하도 오래걸리니까 (약 1분~1분30초) 멈춘건지 작동을 하는건지 가늠이 안가서 작성해줬는데, 그냥 둬도 괜찮아보여서 그냥 뒀습니다.

그리고 마지막까지 MF를 한 결과 나온 b_U, b_I, U, I matrix를 활용해서, 최종 예측 matrix를 만들고, 이를 return 해 줬습니다.

Check_rmse 함수

```
from sklearn.metrics import mean_squared_error


def check_rmse(R, U, I, not_nans, b_U, b_I, mean):
    pred_R = mean + b_U[:, np.newaxis] + b_I[np.newaxis, :] + np.dot(U, I.T)

    users = [not_nan[0] for not_nan in not_nans]
    items = [not_nan[1] for not_nan in not_nans]
    R_not_nans = R[users, items]
    pred_R_not_nans = pred_R[users, items]

    mse = mean_squared_error(R_not_nans, pred_R_not_nans)
    rmse = np.sqrt(mse)

    return rmse
```

앞에서 말했듯이, base 데이터에 적혀있던 rating과 예측 matrix 간에 성능이 어느정도 되는지 확인하는 함수입니다.

매개변수로 받은 변수들을 이용해서 실제 rating과 예측 rating을 비교해보고, sklearn에 존재하는, mean_squared_error 함수를 import해서 mse를 구하고, 루트를 씌워서 rmse를 알아봐줍니다.

Main 함수로 다시 돌아와서...

```
print("output_write_start")
output = open('./data/' + base_name + "_prediction.txt", 'w', encoding='utf-8')
for t in test_data.values:
    predict = cf_result.loc[t[0], t[1]]
    output.write(str(t[0]) + '\t')
    output.write(str(t[1]) + '\t')
    output.write(str(predict) + '\n')
output.close()
print("output_write_end")
return 0
```

MF한 결과 나온 matrix로 다시 table을 만들어주고 test 파일에 있는 user와 item 쌍의 예측 rating 점수를 작성하며 output file을 만들어줍니다.

여기까지 설명을 보시면, SDG_matrix_factorization을 할 때, zero injection을 하는 것도 알 수 있는데, 왜 없냐 하면, PA4.exe에서 zero injection을 사용해서 MF를 했을 경우 rmse가 0.4 정도나 대폭 상승했기 때문입니다. 그리고, zero injection을 하기 위한 MF 과정또한 시간이 추가로 필요하게 되어서 프로그램의 성능(시간)이 훨씬 느려져서 그냥 제거했습니다.

```
zero_one_data = pd.merge(base_data, test_data, how='outer', on=['user_id', 'item_id', 'rating'])
rating = zero_one_data.groupby("rating").count().iloc[:, :1] / zero_one_data.count().user_id
zero_one_table = data.pivot_table('rating', index='user_id', columns='item_id').replace(rating[1:].index, 1).replace(0, np.nan)
zot_matrix = zero_one_table.values

print("zero_injection_start")
zero_injection_pred_matrix = after_matrix_factorization(zot_matrix, True)
zi_result = pd.DataFrame(zero_injection_pred_matrix, columns=rating_table.columns, index=rating_table.index)
print("zero_injection_end")

print("rating_table_change_start")
zero_injection_array = zero_injection_pred_matrix.reshape(-1)
# zero injection의 세타값을 50퍼센트로 잡을 경우이다.
zi_median = np.median(zero_injection_array)
for i in range(len(zero_injection_pred_matrix)):
    for j in range(len(zero_injection_pred_matrix[i])):
        if zero_injection_pred_matrix[i][j] < zi_median:
            rt_matrix[i, j] = 2
print("rating_table_change_end")
```

일단 시도해봤었다는 코드입니다. 하지만 결론적으로 rmse 성능이 오히려 안좋아져서 삭제했습니다.

ü Instructions for compiling your source codes at TA's computer (e.g. screenshot) (Important!!)

```
C:₩workspace₩2022_ite4005_2017030464₩recommender>python recommender.py u1.base u1.test
cf_start
1 times MF's rmse: 0.9742591434891142
2 times MF's rmse: 0.944455679863557
3 times MF's rmse: 0.9303427781597919
4 times MF's rmse: 0.9217068202080794
5 times MF's rmse: 0.915624387391922
6 times MF's rmse: 0.9108681466635515
7 times MF's rmse: 0.9068276757774788
8 times MF's rmse: 0.9031676327732147
9 times MF's rmse: 0.8996927411159902
10 times MF's rmse: 0.8962874502251712
11 times MF's rmse: 0.892886884346106
12 times MF's rmse: 0.8894618397262317
13 times MF's rmse: 0.8860096913955489
14 times MF's rmse: 0.8825470019154713
15 times MF's rmse: 0.8791021354547222
16 times MF's rmse: 0.8757080746540916
17 times MF's rmse: 0.8723965325803417
18 times MF's rmse: 0.8691941773158346
19 times MF's rmse: 0.8661209613609381
20 times MF's rmse: 0.863189970275647
21 times MF's rmse: 0.8604081274884289
22 times MF's rmse: 0.8577772825804769
23 times MF's rmse: 0.8552954057307683
24 times MF's rmse: 0.8529577272806802
25 times MF's rmse: 0.850757725414206
26 times MF's rmse: 0.8486879119838326
27 times MF's rmse: 0.8467404073037729
28 times MF's rmse: 0.8449073257059977
29 times MF's rmse: 0.8431810108249567
30 times MF's rmse: 0.8415541636114514
31 times MF's rmse: 0.8400199010980388
32 times MF's rmse: 0.8385717747939077
33 times MF's rmse: 0.8372037680133472
34 times MF's rmse: 0.8359102835204839
35 times MF's rmse: 0.8346861272960014
36 times MF's rmse: 0.8335264908372064
37 times MF's rmse: 0.8324269326579158
38 times MF's rmse: 0.8313833589715373
39 times MF's rmse: 0.8303920034350397
40 times MF's rmse: 0.8294494059698817
41 times MF's rmse: 0.8285523908636482
MF was performed 41 times.
cf_end
output_write_start
output_write_end

C:₩workspace₩2022_ite4005_2017030464₩recommender>
```

```
C:₩workspace₩2022_ite4005_2017030464₩recommender>python recommender.py u2.base u2.test
cf_start
1 times MF's rmse: 0.9773532122127696
2 times MF's rmse: 0.9473155862184258
3 times MF's rmse: 0.9328453982047183
4 times MF's rmse: 0.923959957211668
5 times MF's rmse: 0.9177272485790353
6 times MF's rmse: 0.9128964004212633
7 times MF's rmse: 0.9088372736293883
8 times MF's rmse: 0.9051997656737246
9 times MF's rmse: 0.9017768165402223
10 times MF's rmse: 0.8984417239270905
11 times MF's rmse: 0.895117211612959
12 times MF's rmse: 0.891759648421965
13 times MF's rmse: 0.8883508134632263
14 times MF's rmse: 0.8848929410771006
15 times MF's rmse: 0.881404327421292
16 times MF's rmse: 0.8779141347469741
17 times MF's rmse: 0.8744564926185925
18 times MF's rmse: 0.8710650150007793
19 times MF's rmse: 0.8677688894989086
20 times MF's rmse: 0.8645909556145497
21 times MF's rmse: 0.8615474071228852
22 times MF's rmse: 0.8586484337649474
23 times MF's rmse: 0.8558992236924235
24 times MF's rmse: 0.8533009964545047
25 times MF's rmse: 0.8508519301862386
26 times MF's rmse: 0.8485479464047228
27 times MF's rmse: 0.8463833547813483
28 times MF's rmse: 0.8443513731073916
29 times MF's rmse: 0.842444454129233
30 times MF's rmse: 0.8406550479883294
31 times MF's rmse: 0.8389749860160237
32 times MF's rmse: 0.83739654943046
33 times MF's rmse: 0.8359121819379406
34 times MF's rmse: 0.8345146841108307
35 times MF's rmse: 0.8331972855491812
36 times MF's rmse: 0.8319536875433147
37 times MF's rmse: 0.8307780815334167
38 times MF's rmse: 0.8296651484454376
39 times MF's rmse: 0.8286100436482914
40 times MF's rmse: 0.8276083717878083
41 times MF's rmse: 0.8266561551449951
42 times MF's rmse: 0.82574979850477
43 times MF's rmse: 0.8248860528685454
MF was performed 43 times.
cf_end
output_write_start
output_write_end

C:₩workspace₩2022_ite4005_2017030464₩recommender>
```

```
C:₩workspace₩2022_ite4005_2017030464₩recommender>python recommender.py u3.base u3.test
cf_start
1 times MF's rmse: 0.9801610699705015
2 times MF's rmse: 0.9493152838138517
3 times MF's rmse: 0.9343006291344458
4 times MF's rmse: 0.9250053926572975
5 times MF's rmse: 0.9183944131575944
6 times MF's rmse: 0.913159235728051
7 times MF's rmse: 0.9086372192079955
8 times MF's rmse: 0.9044595144735132
9 times MF's rmse: 0.9004116929138847
10 times MF's rmse: 0.8963735550448734
11 times MF's rmse: 0.8922917948609276
12 times MF's rmse: 0.888164738097931
13 times MF's rmse: 0.8840282199814452
14 times MF's rmse: 0.8799387262785343
15 times MF's rmse: 0.8759560705919485
16 times MF's rmse: 0.8721303738911258
17 times MF's rmse: 0.8684959945352072
18 times MF's rmse: 0.8650714358657408
19 times MF's rmse: 0.8618624483021112
20 times MF's rmse: 0.8588659087498896
21 times MF's rmse: 0.8560732066183093
22 times MF's rmse: 0.8534727567824164
23 times MF's rmse: 0.8510516911152164
24 times MF's rmse: 0.848796913578534
25 times MF's rmse: 0.8466957077284584
26 times MF's rmse: 0.8447360496040421
27 times MF's rmse: 0.8429067382799076
28 times MF's rmse: 0.8411974214802945
29 times MF's rmse: 0.8395985666445641
30 times MF's rmse: 0.8381014081941479
31 times MF's rmse: 0.836697888260636
32 times MF's rmse: 0.8353805994331795
33 times MF's rmse: 0.8341427328822435
34 times MF's rmse: 0.8329780324101236
35 times MF's rmse: 0.8318807536936144
36 times MF's rmse: 0.8308456275840845
37 times MF's rmse: 0.8298678263856669
38 times MF's rmse: 0.8289429322766767
39 times MF's rmse: 0.8280669073225341
MF was performed 39 times.
cf_end
output_write_start
output_write_end

C:₩workspace₩2022_ite4005_2017030464₩recommender>
```

```
C:₩workspace₩2022_ite4005_2017030464₩recommender>python recommender.py u4.base u4.test
cf_start
1 times MF's rmse: 0.9815039313465416
2 times MF's rmse: 0.9506391127006734
3 times MF's rmse: 0.9357070963696451
4 times MF's rmse: 0.9265328619523843
5 times MF's rmse: 0.9201046098134681
6 times MF's rmse: 0.9151287557079228
7 times MF's rmse: 0.9109461514519553
8 times MF's rmse: 0.9071830805076485
9 times MF's rmse: 0.9036107695775564
10 times MF's rmse: 0.9000824272260478
11 times MF's rmse: 0.8965045523408173
12 times MF's rmse: 0.8928249975222732
13 times MF's rmse: 0.8890284849202311
14 times MF's rmse: 0.8851330583944363
15 times MF's rmse: 0.8811832196563266
16 times MF's rmse: 0.8772389342000237
17 times MF's rmse: 0.8733630997818093
18 times MF's rmse: 0.8696111979885045
19 times MF's rmse: 0.8660253414737414
20 times MF's rmse: 0.862632650895463
21 times MF's rmse: 0.8594465869469121
22 times MF's rmse: 0.8564697300883837
23 times MF's rmse: 0.8536969101770446
24 times MF's rmse: 0.8511180461093265
25 times MF's rmse: 0.8487204047538118
26 times MF's rmse: 0.8464902206703245
27 times MF's rmse: 0.8444137505786007
28 times MF's rmse: 0.8424778909312621
29 times MF's rmse: 0.8406704900578831
30 times MF's rmse: 0.838980464049537
31 times MF's rmse: 0.8373977963905886
32 times MF's rmse: 0.8359134751357932
33 times MF's rmse: 0.8345194015254123
34 times MF's rmse: 0.8332082902177301
35 times MF's rmse: 0.8319735724235309
36 times MF's rmse: 0.8308093076687418
37 times MF's rmse: 0.8297101065242686
38 times MF's rmse: 0.8286710646328793
39 times MF's rmse: 0.8276877072303946
40 times MF's rmse: 0.8267559427922592
41 times MF's rmse: 0.8258720242394357
MF was performed 41 times.
cf_end
output_write_start
output_write_end

C:₩workspace₩2022_ite4005_2017030464₩recommender>
```

```
C:₩workspace₩2022_ite4005_2017030464₩recommender>python recommender.py u5.base u5.test
cf_start
1 times MF's rmse: 0.9810292111506812
2 times MF's rmse: 0.950779892342913
3 times MF's rmse: 0.9361716725238235
4 times MF's rmse: 0.9271434668356344
5 times MF's rmse: 0.9207460730359108
6 times MF's rmse: 0.9157329022674352
7 times MF's rmse: 0.9114837858138805
8 times MF's rmse: 0.9076615900035889
9 times MF's rmse: 0.9040747909264241
10 times MF's rmse: 0.9006139530618866
11 times MF's rmse: 0.8972187195331731
12 times MF's rmse: 0.8938587916445371
13 times MF's rmse: 0.8905220841281433
14 times MF's rmse: 0.8872070987853922
15 times MF's rmse: 0.8839181168272171
16 times MF's rmse: 0.880662404070684
17 times MF's rmse: 0.8774488198770277
18 times MF's rmse: 0.8742872797117649
19 times MF's rmse: 0.8711885775467443
20 times MF's rmse: 0.8681641929240845
21 times MF's rmse: 0.8652258842560998
22 times MF's rmse: 0.8623850513607048
23 times MF's rmse: 0.859651979486104
24 times MF's rmse: 0.857035128170086
25 times MF's rmse: 0.8545406093022735
26 times MF's rmse: 0.8521719371054726
27 times MF's rmse: 0.8499300596000694
28 times MF's rmse: 0.8478136213297065
29 times MF's rmse: 0.8458193741742622
30 times MF's rmse: 0.8439426482963113
31 times MF's rmse: 0.8421778110835183
32 times MF's rmse: 0.840518667279493
33 times MF's rmse: 0.8389587786663828
34 times MF's rmse: 0.8374917009577755
35 times MF's rmse: 0.8361111472795887
36 times MF's rmse: 0.8348110927230699
37 times MF's rmse: 0.8335858350132936
38 times MF's rmse: 0.8324300244153218
39 times MF's rmse: 0.8313386731510976
40 times MF's rmse: 0.8303071517679004
41 times MF's rmse: 0.8293311775348944
42 times MF's rmse: 0.8284067981756952
43 times MF's rmse: 0.827530373025379
MF was performed 43 times.
cf_end
output_write_start
output_write_end

C:₩workspace₩2022_ite4005_2017030464₩recommender>
```

파이썬 파일이라서 cmd 창에서 실행파일이 있는 폴더까지 들어가서

python recommender.py u#.base u#.test

으로 명령어를 치시면 될 것 같습니다.

명세에서 요구한 command는 실행폴더의 command.txt에 정리해뒀습니다.

한줄씩 복붙해서 사용하시면 될 것 같습니다.

ü Any other specification of your implementation and testing

```
C:₩workspace₩2022_ite4005_2017030464₩recommender₩test>PA4.exe u1
the number of ratings that didn't be predicted: 0
the number of ratings that were unproperly predicted [ex. >=10, <0, NaN, or format errors]: 10
If the counted number is large, please check your codes again.

The bigger value means that the ratings are predicted more incorrectly
RMSE: 0.9587826

C:₩workspace₩2022_ite4005_2017030464₩recommender₩test>PA4.exe u2
the number of ratings that didn't be predicted: 0
the number of ratings that were unproperly predicted [ex. >=10, <0, NaN, or format errors]: 1
If the counted number is large, please check your codes again.

The bigger value means that the ratings are predicted more incorrectly
RMSE: 0.9438544

C:₩workspace₩2022_ite4005_2017030464₩recommender₩test>PA4.exe u3
the number of ratings that didn't be predicted: 0
the number of ratings that were unproperly predicted [ex. >=10, <0, NaN, or format errors]: 3
If the counted number is large, please check your codes again.

The bigger value means that the ratings are predicted more incorrectly
RMSE: 0.9483804

C:₩workspace₩2022_ite4005_2017030464₩recommender₩test>PA4.exe u4
the number of ratings that didn't be predicted: 0
the number of ratings that were unproperly predicted [ex. >=10, <0, NaN, or format errors]: 3
If the counted number is large, please check your codes again.

The bigger value means that the ratings are predicted more incorrectly
RMSE: 0.9425129

C:₩workspace₩2022_ite4005_2017030464₩recommender₩test>PA4.exe u5
the number of ratings that didn't be predicted: 0
the number of ratings that were unproperly predicted [ex. >=10, <0, NaN, or format errors]: 2
If the counted number is large, please check your codes again.

The bigger value means that the ratings are predicted more incorrectly
RMSE: 0.946911

C:₩workspace₩2022_ite4005_2017030464₩recommender₩test>
```

Base 와 test 파일은 recommender 폴더의 data 폴더에 넣어주시고,
PA4.exe 파일을 이용하여 test 를 원하신다면, recommender.py 실행결과 나온 output 파일들을 test 폴더에 test 파일과 함께 넣어서 명세에 적힌대로 비교를 하시면 됩니다.

제 코드의 경우 rating 을 하지못한 값은 없지만, 적정 범위내의 예측에 실패하여 적정한 값이 들어가있지 않는 것이 몇 개 있는데, 확인해본 결과 -0.xxx 의 값이였습니다.

Piazza 에 질문을 올리긴 했으나, 답변이 아직(2022-06-18/오전 8:53) 달리지 않아서 이 unproperly predicted 의 개수를 줄이는 것이 RMSE 를 줄이는 것보다 중요한 것인지 확인하지 못하여 코드를 수정하지 못했습니다.

하지만 여러 번의 실험과 테스트 결과 겨우 10 개 내지의 개수를 적정범위의 수로 바꾼다고 해도 RMSE 에는 큰 변화가 없을 것임에 확신합니다.

지금 한가지 생각난 방법을 작성해보자면, MF 가 끝난 후, 0 보다 작은 값이나, 실제 rating 범위에 맞지 않는 수를 ex) 1~5 범위라면 / 1 보다 작은 것들은 1 로바꾸고, 5 보다 큰 것들은 5 로 바꾸어서 output 파일에 예측값으로 적어준다면, 당연히 RMSE 는 더 줄어들 것이라고 확신하지만, 이 방법도 전에 질문했던, 예측값을 반올림하여 정수로 만들어서 작성해야하는지 piazza 에 질문한 결과 하지 않아야한다고 하셨던 것을 생각해보면, 이 방법또한 높은확률로 안될 것 같습니다만 우선 piazza 에 질문을 작성해놨습니다.