Grading

The final score that you will receive for your programming assignment is generated in relation to the total points set in your programming assignment item—not the total point value in the nbgrader notebook. When calculating the final score shown to learners, the programming assignment takes the percentage of earned points vs. the total points provided by nbgrader and returns a score matching the equivalent percentage of the point value for the programming assignment.
DO NOT CHANGE VARIABLE OR METHOD SIGNATURES The autograder will not work properly if your change the variable or method signatures.

Validate Button

Please note that this assignment uses nbgrader to facilitate grading. You will see a **validate button** at the top of your Jupyter notebook. If you hit this button, it will run tests cases for the lab that aren't hidden. It is good to use the validate button before submitting the lab. Do know that the labs in the course contain hidden test cases. The validate button will not let you know whether these test cases pass. After submitting your lab, you can see more information about these hidden test cases in the Grader Output.
Cells with longer execution times will cause the validate button to time out and freeze. Please know that if you run into Validate time-outs, it will not affect the final submission grading.

Building Recommender Systems for Movie Rating Prediction

In this assignment, we will build a recommender systems that predict movie ratings. [MovieLense (https://grouplens.org/datasets/movielens/)](https://grouplens.org/datasets/movielens/) has currently 25 million user-movie ratings. Since the entire data is too big, we use a 1 million ratings subset [MovieLens 1M (https://www.kaggle.com/odedgolden/movielens-1m-dataset)](https://www.kaggle.com/odedgolden/movielens-1m-dataset), and we reformatted the data to make it more convenient to use.

In [1]:
```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import time
from sklearn.model_selection import train_test_split
from scipy.sparse import coo_matrix, csr_matrix
from scipy.spatial.distance import jaccard, cosine
from pytest import approx
```

In [2]:
```
MV_users = pd.read_csv('data/users.csv')
MV_movies = pd.read_csv('data/movies.csv')
train = pd.read_csv('data/train.csv')
test = pd.read_csv('data/test.csv')
```

In [3]:
```
from collections import namedtuple
Data = namedtuple('Data', ['users','movies','train','test'])
data = Data(MV_users, MV_movies, train, test)
```

Starter codes

Now, we will be building a recommender system which has various techniques to predict ratings. The `class RecSys` has baseline prediction methods (such as predicting everything to 3 or to average rating of each user) and other utility functions. `class ContentBased` and `class Collaborative` inherit `class RecSys` and further add methods calculating item-item similarity matrix. You will be completing those functions using what we learned about content-based filtering and collaborative filtering.

`RecSys` 's `rating_matrix` method converts the (user id, movie id, rating) triplet from the train data (train data's ratings are known) into a utility matrix for 6040 users and 3883 movies.
Here, we create the utility matrix as a dense matrix (numpy.array) format for convenience. But in a real world data where hundreds of millions of users and items may exist, we won't be able to create the utility matrix in a dense matrix format (For those who are curious why, try measuring the dense matrix self.Mr using .nbytes()). In that case, we may use sparse matrix operations as much as possible and distributed file systems and distributed computing will be needed. Fortunately, our data is small enough to fit in a laptop/pc memory. Also, we will use numpy and scipy.sparse, which allow significantly faster calculations than calculating on pandas.DataFrame object.
In the `rating_matrix` method, pay attention to the index mapping as user IDs and movie IDs are not the same as array index.

In [4]:
```
class RecSys():
    def __init__(self,data):
        self.data=data
        self.allusers = list(self.data.users['uID'])
        self.allmovies = list(self.data.movies['mID'])
        self.genres = list(self.data.movies.columns.drop(['mID', 'title',
'year']))
        self.mid2idx = dict(zip(self.data.movies.mID,list(range(len(sel
f.data.movies)))))
        self.uid2idx = dict(zip(self.data.users.uID,list(range(len(self.d
ata.users)))))
        self.Mr=self.rating_matrix()
        self.Mm=None
        self.sim=np.zeros((len(self.allmovies),len(self.allmovies)))

    def rating_matrix(self):
        ind_movie = [self.mid2idx[x] for x in self.data.train.mID]
        ind_user = [self.uid2idx[x] for x in self.data.train.uID]
        rating_train = list(self.data.train.rating)
        return np.array(coo_matrix((rating_train, (ind_user, ind_movie)),
shape=(len(self.allusers), len(self.allmovies))).todense())

    def predict_everything_to_3(self):
        return np.full(self.data.test.shape[0], 3)

    def predict_to_user_average(self):

        mean_ratings = dict()
        test_IDs = self.data.test.uID.unique()
        for id in test_IDs:
            id_idx = self.uid2idx[id]
            ratings = self.Mr[id_idx]
            mean_ratings[id] = ratings.sum() / np.count_nonzero(ratings)

        yp = []
        for i in range(len(self.data.test)):
            yp.append(mean_ratings[self.data.test.uID[i]])
        yp = np.array(yp)
        return yp

    def predict_from_sim(self, uid, mid):
        index_userID = self.uid2idx[uid]
        ratings_index_userID = self.Mr[index_userID]
        index_movieID = self.mid2idx[mid]
        movie_sims = self.sim[index_movieID]
        sum_of_sims = np.dot(movie_sims, ratings_index_userID !=0)  # sum
of sims where rating != 0
        rating = np.dot(ratings_index_userID, movie_sims) / sum_of_sims

        # if there are no similar movies, ie all sims=0 then the rating w
ill be 0
        # if rating=0 then predict to user average
        if rating == 0:
            return self.Mr[index_userID].sum() / np.count_nonzero(self.Mr
[index_userID])
        else:
            return rating

    def predict(self):
```

```
        yp = np.array([])
        for i in range(len(self.data.test)):
            uID = self.data.test.iloc[i]['uID']
            mID = self.data.test.iloc[i]['mID']
            rating = self.predict_from_sim(uID, mID)
            yp = np.append(yp, rating)
        return yp

    def rmse(self, yp):
        yp[np.isnan(yp)] = 3 #In case there is nan values in prediction, it will impute to 3.
        yt=np.array(self.data.test.rating)
        return np.sqrt(((yt-yp)**2).mean())


class ContentBased(RecSys):
    def __init__(self,data):
        super().__init__(data)
        self.data=data
        self.Mm = self.calc_movie_feature_matrix()

    def calc_movie_feature_matrix(self):
        """
        Create movie feature matrix in a numpy array of shape (#allmovies, #genres)
        """
        # your code here

        return np.array(self.data.movies[self.genres])

    def calc_item_item_similarity(self):
        """
        Create item-item similarity using Jaccard similarity
        """
        # Update the sim matrix by calculating item-item similarity using Jaccard similarity
        # Jaccard Similarity: J(A, B) = |A∩B| / |A∪B|
        # your code here
        from sklearn.metrics import pairwise_distances
        self.sim = (1 - pairwise_distances(np.array(self.Mm), metric='jaccard'))
        return

class Collaborative(RecSys):
    def __init__(self,data):
        super().__init__(data)

    def calc_item_item_similarity(self, simfunction, *X):
        """
        Create item-item similarity using similarity function.
        X is an optional transformed matrix of Mr
        """
        # General function that calculates item-item similarity based on the sim function and data inputed
        if len(X)==0:
            self.sim = simfunction()
        else:
            self.sim = simfunction(X[0]) # *X passes in a tuple format of (X,), to X[0] will be the actual transformed matrix
```

```
    def cossim(self):
        """
        Calculates item-item similarity for all pairs of items using cosi
ne similarity (values from 0 to 1) on utility matrix
        Returns a cosine similarity matrix of size (#all movies, #all mov
ies)
        """
        # Return a sim matrix by calculating item-item similarity for all
pairs of items using Jaccard similarity
        # Cosine Similarity: C(A, B) = (A.B) / (||A||.||B||)
        # your code here

        # Compute **averaged** movie ratings for all users (movie_ratings
_allUsers)
        movie_ratings_allUsers = self.Mr.sum(axis=1) / np.count_nonzero(s
elf.Mr, axis=1)
        np.nan_to_num(movie_ratings_allUsers, copy=False)  # default copy
=True

        # Create a sparse matrix for operating cosine on its values
        movie_ratings_array = np.repeat(np.expand_dims(movie_ratings_allU
sers, axis=1), self.Mr.shape[1], axis=1)

        # Take care of all the zero ratings (missing value/itentionally w
e don't know)
        movie_ratings_array_adjusted = self.Mr + (self.Mr==0)*movie_ratin
gs_array - movie_ratings_array

        # Average all the ratings: divide by its magnitude!
        MR_avg = movie_ratings_array_adjusted / (np.sqrt((movie_ratings_a
rray_adjusted**2).sum(axis=0)))

        # Put a Boundary check # 1: since dividing by magnitude may produ
ce inf, zeros, etc. Set nans to 0.
        MR_avg = np.nan_to_num(MR_avg)  # or np.nan_to_num(MR_avg, copy=F
alse)

        # Perform an item-item cosine similarity using: np.dot(matrix.T,
matrix)
        sim_mat = np.dot(MR_avg.T, MR_avg)

        # Note that the 289 movies with all zero rating will have cosine
sim = 0  -  all same-same movie ratings along diagonal should be 1
        #a = np.argwhere(np.diag(sim_mat) == 0)
        #sim_mat[a, a] = 1
        # still 42 (other vals along diagonal slightly > 1) - use alt met
hod
        idx = range(sim_mat.shape[0])
        sim_mat[idx, idx] = 1

        # Normalized Cosine Formula:
        sim_mat = 0.5 + (0.5 * sim_mat)

        return sim_mat

    def jacsim(self,Xr):
        """
        Calculates item-item similarity for all pairs of items using jacc
ard similarity (values from 0 to 1)
        Xr is the transformed rating matrix.
```

```
        """
        # Return a sim matrix by calculating item-item similarity for all
pairs of items using Jaccard similarity
        # Jaccard Similarity: J(A, B) = |A∩B| / |A∪B|
        # your code here

        n = Xr.shape[1]
        max_val = int(Xr.max())
        nz_intersect = np.zeros((n,n)).astype(int)
        for i in range(1, max_val + 1):
                csrm = csr_matrix((Xr == i)).astype(int)
                nz_intersect = nz_intersect + np.array(np.dot(csrm.T, csr
m).toarray()).astype(int)

        # get the union

        # get the nonzero counts of each column
        #colsums = A.sum(axis=0)  # alternatively
        colsums = np.count_nonzero(Xr, axis=0)  # alternatively

        # get matrix of sum of colsums between columns
        # start with matrix of n x n where row vals = sum for correspondi
ng column eg col 1 = 4, all row[0] vals = 4
        n = Xr.shape[1] # how many movies / columns
        colsums_mat = np.repeat(colsums.reshape(n,1), n, axis=1)
        # add the colsum matrix to its transpose to get the pairs
        colsums_pairs = colsums_mat + colsums_mat.T

        # to get the union:  subtract the intersection of a pair from the
column sums of the two colums  eg col 1 = 4, col 2 = 3; total = 7, int =
3 ---> untion = 4
        union = colsums_pairs - nz_intersect

        # calculate jaccard similarity
        sim = nz_intersect / union
        np.nan_to_num(sim, copy=False)  # NaNs potentially generated when
union is zero

        d = np.argwhere(np.diag(sim) != 1)
        sim[d, d] = 1

        return np.array(sim)
```

Q1. Baseline models [15 pts]

1a. Complete the function `predict_everything_to_3` in the class `RecSys` [5 pts]

In [5]:
```
# Creating Sample test data
np.random.seed(42)
sample_train = train[:30000]
sample_test = test[:30000]


sample_MV_users = MV_users[(MV_users.uID.isin(sample_train.uID)) | (MV_users.uID.isin(sample_test.uID))]
sample_MV_movies = MV_movies[(MV_movies.mID.isin(sample_train.mID)) | (MV_movies.mID.isin(sample_test.mID))]


sample_data = Data(sample_MV_users, sample_MV_movies, sample_train, sample_test)
```

In [6]:
```
# Sample tests predict_everything_to_3 in class RecSys

sample_rs = RecSys(sample_data)
sample_yp = sample_rs.predict_everything_to_3()
print(sample_rs.rmse(sample_yp))
assert sample_rs.rmse(sample_yp)==approx(1.2642784503423288, abs=1e-3), "Did you predict everything to 3 for the test data?"
```

```
1.2642784503423288
```

In [7]:
```
# Hidden tests predict_everything_to_3 in class RecSys
rs = RecSys(data)
yp = rs.predict_everything_to_3()
print(rs.rmse(yp))
```

```
1.2585510334053043
```

1b. Complete the function predict_to_user_average in the class RecSys [10 pts]

Hint: Include rated items only when averaging

In [ ]:
```
# Sample tests predict_to_user_average in the class RecSys
sample_yp = sample_rs.predict_to_user_average()
print(sample_rs.rmse(sample_yp))
assert sample_rs.rmse(sample_yp)==approx(1.1429596846619763, abs=1e-3), "Check predict_to_user_average in the RecSys class. Did you predict to average rating for the user?"
```

In [ ]:
```
print(sample_rs.rmse(sample_yp))
```

In [8]:
```
# Hidden tests predict_to_user_average in the class RecSys
yp = rs.predict_to_user_average()
print(rs.rmse(yp))
```

```
1.0352910334228647
```

Q2. Content-Based model [25 pts]

2a. Complete the function calc_movie_feature_matrix in the class ContentBased [5 pts]

In [9]:
```
cb = ContentBased(data)
```

In [10]:
```
# tests calc_movie_feature_matrix in the class ContentBased
assert(cb.Mm.shape==(3883, 18))
```

2b. Complete the function calc_item_item_similarity in the class ContentBased [10 pts]

This function updates `self.sim` and does not return a value.
Some factors to think about:

1. The movie feature matrix has binary elements. Which similarity metric should be used?
2. What is the computation complexity (time complexity) on similarity calcuation?
   Hint: You may use functions in the `scipy.spatial.distance` module on the dense matrix, but it is quite slow (think about the time complexity). If you want to speed up, you may try using functions in the `scipy.sparse` module.

In [11]:
```
cb.calc_item_item_similarity()
```

In [12]:
```
# Sample tests calc_item_item_similarity in ContentBased class

sample_cb = ContentBased(sample_data)
sample_cb.calc_item_item_similarity()

# print(np.trace(sample_cb.sim))
# print(sample_cb.sim[10:13,10:13])
assert(sample_cb.sim.sum() > 0), "Check calc_item_item_similarity."
assert(np.trace(sample_cb.sim) == 3152), "Check calc_item_item_similarity
. What do you think np.trace(cb.sim) should be?"


ans = np.array([[1, 0.25, 0.],[0.25, 1, 0.],[0., 0., 1]])
for pred, true in zip(sample_cb.sim[10:13, 10:13], ans):
    assert approx(pred, 0.01) == true, "Check calc_item_item_similarity.
Look at cb.sim"
```

In [13]:
```
# tests calc_item_item_similarity in ContentBased class
```

In [14]:
```
# additional tests for calc_item_item_similarity in ContentBased class
```

In [15]:
```
# additional tests for calc_item_item_similarity in ContentBased class
```

In [16]:
```
# additional tests for calc_item_item_similarity in ContentBased class
```

In [17]:
```
# additional tests for calc_item_item_similarity in ContentBased class
```

2c. Complete the function predict_from_sim in the class RecSys [5 pts]

In [18]:
```
# for a, b in zip(sample_MV_users.uID, sample_MV_movies.mID):
#     print(a, b, sample_cb.predict_from_sim(a,b))

# Sample tests for predict_from_sim in RecSys class
assert(sample_cb.predict_from_sim(245,276)==approx(2.5128205128205128,abs
=1e-2)), "Check predict_from_sim. Look at how you predicted a user rating
on a movie given UserID and movieID."
assert(sample_cb.predict_from_sim(2026,2436)==approx(2.785714285714286,ab
s=1e-2)), "Check predict_from_sim. Look at how you predicted a user ratin
g on a movie given UserID and movieID."
```

In [19]:
```
# tests for predict_from_sim in RecSys class
```

2d. Complete the function predict in the class RecSys [5 pts]

After completing the predict method in the RecSys class, run the cell below to calculate rating prediction and RMSE. How much does the performance increase compared to the baseline results from above?

In [20]:
```
# Sample tests method predict in the RecSys class

sample_yp = sample_cb.predict()
sample_rmse = sample_cb.rmse(sample_yp)
print(sample_rmse)

assert(sample_rmse==approx(1.1962537249116723, abs=1e-2)), "Check method
predict in the RecSys class."
```

```
1.1962537249116723
```

In [21]:
```
# Hidden tests method predict in the RecSys class

yp = cb.predict()
rmse = cb.rmse(yp)
print(rmse)
```

```
1.0128116783754684
```

In [22]:
```
# tests method predict in the RecSys class
```

Q3. Collaborative Filtering

3a. Complete the function cossim in the class Collaborative [10 pts]

To Do:
1.Impute the unrated entries in self.Mr to the user's average rating then subtract by the user mean, call this matrix X.
2.Calculate cosine similarity for all item-item pairs. Don't forget to rescale the cosine similarity to be 0~1. You might encounter divide by zero warning (numpy will fill nan value for that entry). In that case, you can fill those with appropriate values.

Hint: Let's say a movie item has not been rated by anyone. When you calculate similarity of this vector to anoter, you will get $\vec{0}$=[0,0,0,....,0]. When you normalize this vector, you'll get divide by zero warning and it will make nan value in self.sim matrix. Theoretically what should the similarity value for $\vec{x}_i \cdot \vec{x}_i$ when $\vec{x}_i = \vec{0}$? What about $\vec{x}_i \cdot \vec{x}_j$ when $\vec{x}_i = \vec{0}$ and $\vec{x}_j$ is an any vector?

Hint: You may use `scipy.spatial.distance.cosine` , but it will be slow because its cosine function does vector-vector operation whereas you can implement matrix-matrix operation using numpy to calculate all cosines all at once (it can be 100 times faster than vector-vector operation in our data). Also pay attention to the definition. The scipy.spatial.distance provides distance, not similarity.

1. Run the below cell that calculate yp and RMSE.

In [23]:
```
# Sample tests cossim method in the Collaborative class

sample_cf = Collaborative(sample_data)
sample_cf.calc_item_item_similarity(sample_cf.cossim)
sample_yp = sample_cf.predict()
sample_rmse = sample_cf.rmse(sample_yp)

assert(np.trace(sample_cf.sim)==3152), "Check cossim method in the Collab
orative class. What should np.trace(cf.sim) equal?"
assert(sample_rmse==approx(1.1429596846619763, abs=5e-3)), "Check cossim
method in the Collaborative class. rmse result is not as expected."
assert(sample_cf.sim[0,:3]==approx([1., 0.5, 0.5],abs=1e-2)), "Check coss
im method in the Collaborative class. cf.sim isn't giving the expected re
sults."
```

In [24]:
```
# Hidden tests cossim method in the Collaborative class

cf = Collaborative(data)
cf.calc_item_item_similarity(cf.cossim)
yp = cf.predict()
rmse = cf.rmse(yp)
print(rmse)
```

```
1.0263081874204125
```

In [25]:
```
# tests cossim method in the Collaborative class
```

In [26]:
```
# additional tests for cossim method in the Collaborative class
```

In [27]:
```
# additional tests for cossim method in the Collaborative class
```

In [28]:
```
# additional tests for cossim method in the Collaborative class
```

In [29]:
```
# additional tests for cossim method in the Collaborative class
```

In [30]:
```
# additional tests for cossim method in the Collaborative class
```

3b. Complete the function jacsim in the class Collaborative [15 pts]

3b [15 pts] = 3b-i) [5 pts]+3b-ii) [5 pts]+ 3b-iii) [5 pts]

Function `jacsim` calculates jaccard similarity between items using collaborative filtering method. When we have a rating matrix `self.Mr` , the entries of Mr matrix are 0 to 5 (0: unrated, 1-5: rating). We are interested to see which threshold method works better when we use jaccard dimilarity in the collaborative filtering.
We may treat any rating 3 or above to be 1 and the negatively rated (below 3) and no-rating as 0. Or, we may treat movies with any ratings to be 1 and ones that has no rating as 0. In this question, we will complete a function jacsim that takes a transformed rating matrix X and calculate and returns a jaccard similarity matrix.
Let's consider these input cases for the utility matrix M_r with ratings 1-5 and 0s for no-rating.

1. $M_r \geq 3$
2. $M_r \geq 0$
3. M_r, no transform.

Things to think about:

- The cases 1 and 2 are straightforward to calculate Jaccard, but what does Jaccard mean for multicategory data?
- Time complexity: The matrix M_r is much bigger than the item feature matrix M_m, therefore it will take very long time if we calculate on dense matrix.
  Hint: Use sparse matrix.
- Which method will give the best performance?

3b-i) When $M_r \geq 3$ [5 pts]

After you've implemented the jacsim function, run the code below. If implemented correctly, you'll have RMSE below 0.99.

In [31]:
```
cf = Collaborative(data)
Xr = cf.Mr>=3
t0=time.perf_counter()
cf.calc_item_item_similarity(cf.jacsim,Xr)
t1=time.perf_counter()
time_sim = t1-t0
print('similarity calculation time',time_sim)
yp = cf.predict()
rmse = cf.rmse(yp)
print(rmse)
assert(rmse<0.99)
```

```
similarity calculation time 1.474940464948304
0.9819058692126349
```

In [32]:
```
# tests RMSE for jacsim implementation
```

In [33]:
```
# additional tests for RMSE for jacsim implementation
```

In [34]:
```
# additional tests for jacsim implementation
```

In [35]:
```
# additional tests for jacsim implementation
```

3b-ii) When $M_r \geq 1$ [5 pts]

After you've implemented the jacsim function, run the code below. If implemented correctly, you'll have RMSE below 1.0.

In [36]:
```
cf = Collaborative(data)
Xr = cf.Mr>=1
t0=time.perf_counter()
cf.calc_item_item_similarity(cf.jacsim,Xr)
t1=time.perf_counter()
time_sim = t1-t0
print('similarity calculation time',time_sim)
yp = cf.predict()
rmse = cf.rmse(yp)
print(rmse)
assert(rmse<1.0)
```

```
similarity calculation time 1.6456412660190836
0.991363571262366
```

In [37]:
```
# tests RMSE for jacsim implementation
```

In [38]:
```
# tests RMSE for jacsim implementation
```

In [39]:
```
# tests jacsim implementation
```

In [40]:
```
# tests performance of jacsim implementation
```

3b-iii) When M_r; no transform [5 pts]

After you've implemented the jacsim function, run the code below. If implemented correctly, you'll have RMSE below 0.96

In [41]:
```
cf = Collaborative(data)
Xr = cf.Mr.astype(int)
t0=time.perf_counter()
cf.calc_item_item_similarity(cf.jacsim,Xr)
t1=time.perf_counter()
time_sim = t1-t0
print('similarity calculation time',time_sim)
yp = cf.predict()
rmse = cf.rmse(yp)
print(rmse)
assert(rmse<0.96)
```

```
similarity calculation time 2.5726552279666066
0.9516534264490534
```

In [42]:
```
# tests jacsim implementation RMSE
```

In [43]:
```
# tests jacsim implementation RMSE
```

In [44]:
```
# tests jacsim implementation
```

In [45]:
```
# tests jacsim implementation performance
```

3.C Discussion [Peer Review]

Answer the questions below in this week's Peer Review assignment.

1. Summarize the methods and performances: Below is a template/example.

Method	RMSE
Baseline, Y_p=3	
Baseline, $Y_p = \mu_u$	
Content based, item-item	
Collaborative, cosine	
Collaborative, jaccard, $M_r \geq 3$	
Collaborative, jaccard, $M_r \geq 1$	
Collaborative, jaccard, M_r	

1. Discuss which method(s) work better than others and why.

Method	**RMSE**
Baseline, Y_p=3	1.258
Baseline, $Y_p = \mu_u$	1.035
Content based, item-item	1.012
Collaborative, cosine	1.026
Collaborative, jaccard, $M_r \geq 3$	0.9819
Collaborative, jaccard, $M_r \geq 1$	0.9913
Collaborative, jaccard, M_r	0.9516

Based on the provided Root Mean Square Error (RMSE) values for different recommender system methods, we can analyze which methods perform better and why.

1. **Content-based, item-item (RMSE = 1.012):**

   - This method calculates recommendations based on the similarity between items, typically using features or attributes of items.
   - **Advantages:** It can perform well when there is enough item metadata available and when users' preferences can be reasonably inferred from item features.
   - **Why it works well here:** It achieves a relatively low RMSE compared to other methods, suggesting that the item-item similarity approach is effective in predicting user preferences.
2. **Collaborative filtering, jaccard, $(M_r \geq 3)$ (RMSE = 0.9819):**

   - This method uses a collaborative filtering approach where similarity between users is computed using the Jaccard similarity coefficient, considering only users who have rated at least 3 items in common.
   - **Advantages:** By focusing on users who have more interactions (at least 3 common ratings), it potentially improves the quality of similarity metrics and recommendations.
   - **Why it works well here:** It achieves a lower RMSE compared to other collaborative filtering variations, indicating that this approach captures more meaningful user similarities and improves recommendation accuracy.
3. **Collaborative filtering, jaccard, $(M_r \geq 1)$ (RMSE = 0.9913):**

   - Similar to the previous method but considering users who have rated at least 1 item in common.
   - **Advantages:** It expands the pool of potential neighbors (similar users), which can lead to more recommendations but potentially at the cost of lower accuracy if the similarity measure is less discriminative.
   - **Why it performs slightly worse:** The slightly higher RMSE suggests that including users with fewer common ratings (1 or more) might introduce noise or less reliable similarities compared to the stricter $(M_r \geq 3)$ criterion.
4. **Collaborative filtering, jaccard, (M_r) (RMSE = 0.9516):**

   - This method uses Jaccard similarity without a specific threshold on the number of common ratings (M_r).
   - **Advantages:** It considers all users who have rated at least one item, providing a broader range of potential recommendations.
   - **Why it works best:** It achieves the lowest RMSE among all methods listed, indicating that despite potential noise from users with minimal interactions, the overall approach effectively captures user preferences and improves recommendation accuracy.

Comparison and Conclusion:

- **Content-based item-item** and **Collaborative filtering with Jaccard similarity (general or with $(M_r \geq 3)$)** consistently perform better than other methods in terms of RMSE.
- **Why they work better:** These methods leverage either item features or user interactions (with effective similarity metrics like Jaccard) to make personalized recommendations. They are able to capture nuanced user preferences and item similarities effectively, leading to more accurate predictions.

In summary, **Collaborative filtering with Jaccard similarity**, particularly without strict thresholds on common ratings ((M_r)), tends to work best for this recommender system based on the provided RMSE values. It balances broad coverage of potential recommendations with accurate prediction capabilities, outperforming simpler baselines and other variations of collaborative filtering.

```
In [ ]:
```