A Movie-Based Content Based Filtering

This project aimed to create two types of recommenders. One being a content-based filter and the other being user-user (or collaborative) filters. Collaborative filtering is when a recommendation is made based upon what other people liked whose tastes are similar to your own. Ex. If many people who rated 'Rambo' highly also rated 'The Expendables' highly, recommending to you "The expandables" based on your high rating of 'Rambo'. Content-based filter is based upon recommending movies based of the movies properties which are similar to past content you've liked. Ex. Recommending highly rated 1980s sci-fi films 'Aliens' or 'The Fly' to a retro sci-fi film fanatic.

The source used for this project was as listed below. These researchers in 2015 gathered 22884377 ratings and 586994 tag applications across 34208 movies created by 247753 users between January 09, 1995 and January 29, 2016 from the movie website MovieLens.org. Users were selected at random for inclusion from a pool of users who had rated at least 1 movie.

F. Maxwell Harper and Joseph A. Konstan. 2015. The MovieLens Datasets: History and Context. ACM Transactions on Interactive Intelligent Systems (TiiS) 5, 4, Article 19 (December 2015), 19 pages. DOI=http://dx.doi.org/10.1145/2827872

Table of contents

Acquiring the Data

The source for the data can be acquired from IBM Cloud link below and extracted into the same directory as this Jupyter Notebook to be executed. If using a Linux shell, consider adding !wget and !zip before the url/filename to extract the data. Data can also be manually extracted with any zipping archive program on Windows through file explorer.

In [342…

```
print('https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDevelop
```

https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDeveloperSkillsNetwork-ML0101EN-SkillsNetwork/labs/Module%205/data/moviedataset.zip

---

Preprocessing

Some proprocessing tasks are necessary to perform as part of this project. Including importing necessary packages, loading the data into a dataframe and dropping columns not needed for this analysis.

In [410…

```
#Dataframe manipulation library
import pandas as pd
#Math functions, we'll only need the sqrt function so let's import only that
from math import sqrt
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

This process requires up to 1GB of Ram as the ratings-file extracted is over 600MB and movies file over 100MB plus overhead. Loading into memory from a solid state disk will take about 5-10 seconds. On a mechanical Hard Drive, there may be 1-2 minute load time.

In [455…

```
#Storing the movie information into a pandas dataframe
movies_df = pd.read_csv('movies.csv')
#Storing the user information into a pandas dataframe
ratings_df = pd.read_csv('ratings.csv')
#Head is a function that gets the first N rows of a dataframe. N's default is 5.
movies_df.head()
```

Out[455…

	movieId	title	genres
0	1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy
1	2	Jumanji (1995)	Adventure|Children|Fantasy
2	3	Grumpier Old Men (1995)	Comedy|Romance
3	4	Waiting to Exhale (1995)	Comedy|Drama|Romance
4	5	Father of the Bride Part II (1995)	Comedy

Let's remove the year from the **title** column by using pandas' replace function and store it in a new **year** column.

In [456…

```
#Using regular expressions to find a year stored between parentheses
#We specify the parantheses so we don't conflict with movies that have years in thei
movies_df['year'] = movies_df.title.str.extract('(\(\d\d\d\d\))',expand=False)
#Removing the parentheses
movies_df['year'] = movies_df.year.str.extract('(\d\d\d\d)',expand=False)
#Removing the years from the 'title' column
movies_df['title'] = movies_df.title.str.replace('(\(\d\d\d\d\))', '')
```

```
#Applying the strip function to get rid of any ending whitespace characters that may
movies_df['title'] = movies_df['title'].apply(lambda x: x.strip())
movies_df.head()
```

```
C:\Users\WBurc\AppData\Local\Temp\ipykernel_25144\1143695627.py:7: FutureWarning: Th
e default value of regex will change from True to False in a future version.
  movies_df['title'] = movies_df.title.str.replace('(\(\d\d\d\d\))', '')
```

Out[456…

	movieId	title	genres	year
0	1	Toy Story	Adventure|Animation|Children|Comedy|Fantasy	1995
1	2	Jumanji	Adventure|Children|Fantasy	1995
2	3	Grumpier Old Men	Comedy|Romance	1995
3	4	Waiting to Exhale	Comedy|Drama|Romance	1995
4	5	Father of the Bride Part II	Comedy	1995

Next, as we are doing user-user or collaborate content, we focus on similarity between users not similarity between content. So we can save space by dropping content about our movies from the movies dataframe we will not be using. In this case, we drop "genres".

In [457…

```
#Dropping the genres column
movies_df = movies_df.drop('genres', 1)
```

```
C:\Users\WBurc\AppData\Local\Temp\ipykernel_25144\37422046.py:2: FutureWarning: In a
future version of pandas all arguments of DataFrame.drop except for the argument 'la
bels' will be keyword-only.
  movies_df = movies_df.drop('genres', 1)
```

After our loading, cleaning, transformation steps in our ETL pipeline, this is the result of the movies dataframe. We have 3 columns (movieId, title and year) and 34,208 movie entries.

In [458…

```
movies_df.head()
```

Out[458…

	movieId	title	year
0	1	Toy Story	1995
1	2	Jumanji	1995
2	3	Grumpier Old Men	1995
3	4	Waiting to Exhale	1995
4	5	Father of the Bride Part II	1995

In [459…

```
movies_df.shape
```

Out[459…

```
(34208, 3)
```

The other dataframe, now needs to be cleaned and transformed so we'll make another pipeline. Before cleaning/transformation, the rating dataframe(df) looks like this:

In [460…
```
ratings_df.head()
```

Out[460…

	userId	movieId	rating	timestamp
0	1	169	2.5	1204927694
1	1	2471	3.0	1204927438
2	1	48516	5.0	1204927435
3	2	2571	3.5	1436165433
4	2	109487	4.0	1436165496

Next, examining the ratings dataframe columns via running the head function, we can see some data isn't necessary to our recommender system. The actual "timestamp" or "when" a rating was made is not important. What is important is who rated it, what movie they rated and what rating they gave.

In [461…
```
#Drop removes a specified row or column from a dataframe
ratings_df = ratings_df.drop('timestamp', 1)
```

```
C:\Users\WBurc\AppData\Local\Temp\ipykernel_25144\1971122656.py:2: FutureWarning: In
a future version of pandas all arguments of DataFrame.drop except for the argument
'labels' will be keyword-only.
  ratings_df = ratings_df.drop('timestamp', 1)
```

Here's how the final ratings Dataframe looks like:

In [462…
```
ratings_df.head()
```

Out[462…

	userId	movieId	rating
0	1	169	2.5
1	1	2471	3.0
2	1	48516	5.0
3	2	2571	3.5
4	2	109487	4.0

Collaborative Filtering

Our data is loaded, cleaned and transformed (woohoo!). Now it's time to start our work on collaborative-filtering recommendation system.

As mentioned earlier, this technique differents from content/item-item based filter as it doesn't look at the movie's content itself but rather this technique uses other users thoughts on a movie, by seeking users who rated movies similar to the input user. When the system attempts to find users that have similar preferences and opinions as the input user and then recommends items that they have liked to the input. There are many methods of finding similar users including using machine learning. The one I used =here is going to be based on the *Pearson Correlation Function*.

The process (or algorithm) for creating a User Based recommendation system is as follows:

- Select a sample user with the movies the input user has watched
- Based on his rating of the movies, find the top X neighbours who have the closest rating. Ex. If you rate 'Rambo' 4.5, find as many users who thought 'Rambo' was a 4.5 star film. If there are not X users, find users who rated 'Rambo' either 4 or 5 stars as these two entries are closet "distance" to 4.5 on a number-line mathematically, then 3.5 stars, etc. Continue finding neighbours until you reach "x".
- Get the watched movie record of the user for each of the X neighbours. So for example, if someone rated Rambo 4.5, what else did he rate? Let's say he rated "Predator", "Commando" and "The Expendables" all 3.5 or above. Another person who rated Rambo 4.5, rated "Terminator", "Terminator 2" and "Commando" all 3.5 or above.
- Calculate a similarity score using some formula. A formula might be filter movies rated 3.5 or above and count the number of occurences of the movie. Sort by highest count-first(descending).
- Recommend the items with the highest score. In our hypothetical example, since Commando showed up the most times and had a high rating, maybe it is a good movie to recommend?

To test the system, I've created a sample user input of someone who rates sci-fi/action films highly and children's movies so-so. We'll test the recommendation this system gives to this user:

In [463…

```
userInput = [
            {'title':'Breakfast Club, The', 'rating':3.5},
            {'title':'Toy Story', 'rating':2.5},
            {'title':'Jumanji', 'rating':2},
            {'title':"Pulp Fiction", 'rating':5},
            {'title':'Akira', 'rating':4.5},
            {'title':'Matrix, The', 'rating':5},
            {'title':'Predator','rating':5},
            {'title':'Terminator','rating':5},
            {'title':'Commando','rating':4.5},
            {'title':'Aliens','rating':5},
            {'title':'Alien','rating':4.5}
        ]
inputMovies = pd.DataFrame(userInput)
inputMovies
```

Out[463…

	title	rating

	title	rating
0	Breakfast Club, The	3.5
1	Toy Story	2.5
2	Jumanji	2.0
3	Pulp Fiction	5.0
4	Akira	4.5
5	Matrix, The	5.0
6	Predator	5.0
7	Terminator	5.0
8	Commando	4.5
9	Aliens	5.0
10	Alien	4.5

Add movieId to input user

With the input complete, we want to get the movie's ID using the title. This data is contained within movies dataframe.

We can achieve this by first filtering out the rows that contain the input movie's title and then merging this subset with the input dataframe. We also drop unnecessary columns before we merge the movie titles into the inputMovies dataframe to save memory space.

In [464...

```
#Filtering out the movies by title
inputId = movies_df[movies_df['title'].isin(inputMovies['title'].tolist())]
#Then merging it so we can get the movieId. It's implicitly merging it by title.
inputMovies = pd.merge(inputId, inputMovies)
#Dropping information we won't use from the input dataframe
inputMovies = inputMovies.drop('year', 1)
#Final input dataframe
#If a movie you added in above isn't here, then it might not be in the original
#dataframe or it might spelled differently, please check capitalisation.
inputMovies
```

```
C:\Users\WBurc\AppData\Local\Temp\ipykernel_25144\2035672847.py:6: FutureWarning: In
a future version of pandas all arguments of DataFrame.drop except for the argument
'labels' will be keyword-only.
  inputMovies = inputMovies.drop('year', 1)
```

Out[464...

	movieId	title	rating
0	1	Toy Story	2.5
1	2	Jumanji	2.0
2	296	Pulp Fiction	5.0
3	1200	Aliens	5.0
4	1214	Alien	4.5

	movieId	title	rating
5	1274	Akira	4.5
6	1968	Breakfast Club, The	3.5
7	2571	Matrix, The	5.0
8	3527	Predator	5.0
9	6664	Commando	4.5

Finding the users who has seen the same movies

Now with the movie ID's in our input dataframe, we can get the subset of users that have watched and reviewed the movies in our input dataframe. We'll also check the shape to see how many users and movie ratings we found.

In [465…
```
#Filtering out users that have watched movies that the input has watched and storing
userSubset = ratings_df[ratings_df['movieId'].isin(inputMovies['movieId'].tolist())]
userSubset.head()
```

Out[465…

	userId	movieId	rating
3	2	2571	3.5
19	4	296	4.0
105	4	2571	4.0
195	5	2571	2.5
338	11	1200	3.0

In [466…
```
userSubset.shape
```

Out[466…
```
(341937, 3)
```

We now apply a shorting mechanism called "group by" on the UserId column. This sorts the rows by user ID to see how individual users voted.

In [467…
```
#Groupby creates several sub dataframes where they all have the same value in the co
userSubsetGroup = userSubset.groupby(['userId'])
```

Let's look at a randomly selected user of the userSubsetGroup. Let's say the user with userID=1130.

In [468…
```
userSubsetGroup.get_group(1130)
```

Out[468…

	userId	movieId	rating
104167	1130	1	0.5

	userId	movieId	rating
104168	1130	2	4.0
104214	1130	296	4.0
104332	1130	1200	4.0
104337	1130	1214	4.5
104363	1130	1274	4.5
104443	1130	1968	4.5
104530	1130	2571	2.0
104632	1130	3527	0.5

Some users may have only watched "one" similar film and others may have watched all "8" films our "hypothetical" customer has watched. So we'll sort these the userSubetGroup by the length of the number of movies in the array. The length comes from the *len* command and the *lambda* operator applies the action to each item in the group. User that share the most movies in common with the input have higher priority.

In [469…

```
#Sorting it so users with movie most in common with the input will have priority
userSubsetGroup = sorted(userSubsetGroup,  key=lambda x: len(x[1]), reverse=True)
```

Now let's look at the first user.

In [470…

```
userSubsetGroup[0:3]
```

Out[470…

```
[(815,
        userId  movieId  rating
  73747     815        1     4.5
  73748     815        2     3.0
  73922     815      296     5.0
  74313     815     1200     4.0
  74322     815     1214     3.5
  74362     815     1274     3.0
  74678     815     1968     4.5
  75004     815     2571     3.5
  75368     815     3527     3.5
  76403     815     6664     2.0),
 (1502,
         userId  movieId  rating
  133876    1502        1     4.0
  133877    1502        2     3.5
  133917    1502      296     5.0
  134021    1502     1200     4.5
  134032    1502     1214     5.0
  134058    1502     1274     4.0
  134133    1502     1968     5.0
  134204    1502     2571     5.0
  134289    1502     3527     4.0
  134455    1502     6664     2.5),
 (1599,
```

```
        userId  movieId  rating
142722    1599        1     4.0
142723    1599        2     4.0
142810    1599      296     5.0
143023    1599     1200     5.0
143033    1599     1214     4.5
143068    1599     1274     4.0
143220    1599     1968     4.0
143360    1599     2571     4.5
143513    1599     3527     4.0
143811    1599     6664     4.0)]
```

Similarity of users to input user

Now we test similarity between these users to our inputted user preferences through the *Pearson Correlation Coefficient*. It is used to measure the strength of a linear association between the two variables. The formula for finding this coefficient between sets X and Y with N values can be seen in the image below from Wikipedia.

Pearson correlation is invariant to scaling. It is not effected by multiplying the output (Y) of all elements by a nonzero constant or adding any constant to all elements. For example, if you have two vectors X and Y, then, pearson(X, Y) == pearson(X, 2 * Y + 3). Thus multiplying Y by 2, or the addition of 3, did not change the output of pearson.

$$r = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

The output of Pearson Correlation is a number ranging between -1 to 1. One would be a perfect correlation where Johnny and John love the exact-same movies to the exact same degree and their ratings agreed perfectly. Where as -1, Johnny and Jane, is a perfect negative correlation. Jane absolutely loathes movies Johnny loves and vice-versa; a perfect negative correlation.

The closer to 1, they more similar. The closer to -1, the less similar a user's tastes to the input users.

The Pearson Correlation algorithm is slightly computationally intensive as it scales and the user inputs more users. We also know, we've already sorted the users so the people who watched the same-movies will be at the top of the userSubsetGroup. If people watch identical movies, they might like similar movies. So we can selected a smaller subset like 50 or 100 users to examine their movie choices rather than picking all 35,000+. This limit is imposed because we don't want to waste too much time going through every single user.

In [471…

```
userSubsetGroup = userSubsetGroup[0:100]
```

Now, we calculate the Pearson Correlation between input user and subset group, and store it in a dictionary, where the key is the user Id and the value is the coefficient.

In [472…

```
#Store the Pearson Correlation in a dictionary, where the key is the user Id and the
pearsonCorrelationDict = {}

#For every user group in our subset
for name, group in userSubsetGroup:
    #Let's start by sorting the input and current user group so the values aren't mi
    group = group.sort_values(by='movieId')
    inputMovies = inputMovies.sort_values(by='movieId')
    #Get the N for the formula
    nRatings = len(group)
    #Get the review scores for the movies that they both have in common
    temp_df = inputMovies[inputMovies['movieId'].isin(group['movieId'].tolist())]
    #And then store them in a temporary buffer variable in a list format to facilita
    tempRatingList = temp_df['rating'].tolist()
    #Let's also put the current user group reviews in a list format
    tempGroupList = group['rating'].tolist()
    #Now let's calculate the pearson correlation between two users, so called, x and
    Sxx = sum([i**2 for i in tempRatingList]) - pow(sum(tempRatingList),2)/float(nRa
    Syy = sum([i**2 for i in tempGroupList]) - pow(sum(tempGroupList),2)/float(nRati
    Sxy = sum( i*j for i, j in zip(tempRatingList, tempGroupList)) - sum(tempRatingL

    #If the denominator is different than zero, then divide, else, 0 correlation.
    if Sxx != 0 and Syy != 0:
        pearsonCorrelationDict[name] = Sxy/sqrt(Sxx*Syy)
    else:
        pearsonCorrelationDict[name] = 0
```

By listing the items, we can see the userIDs and pearson correlation ratings. For example, scanning the list, I see user 12921 has a similarty rating of 0.844277838650313 or 84.4% similarity while user 10707 has 93.5% similarity. These people would probably love watching mvies with eachother. By comparison, 4586 has a -0.5528 score or they are fairly opposite in tastes. They might not make good viewing company. If you tried to sort through this list to find the top 3 score, it would take you probably at least two minutes. While a machine can do it miliseconds.

In [473…

```
pearsonCorrelationDict.items()
```

Out[473…

```
dict_items([(815, -0.02556642798308049), (1502, 0.2890155889285442), (1599, 0.547619
0476190511), (1625, 0.4987413913553328), (1950, 0.007436845804221031), (3186, 0.7748
917748917739), (4208, 0.18900682275808595), (4415, 0.03280356017629334), (4586, -0.3
4209490699409134), (6530, 0.24281045302823018), (7235, 0.8102670647257323), (7403,
0.4024016226462459), (8675, 0.48412496383258913), (9663, 0.2989321895781668), (1024
8, -0.2516299559794232), (11769, 0.6201736729460421), (12921, 0.7599632550148199),
(13053, 0.23054221993080123), (14551, 0.10647942749998995), (15137, 0.67101321594512
72), (15157, 0.6936746053755912), (16456, 0.48835287447707093), (17757, 0.2809057653
5151063), (17897, 0.31943828249996986), (19607, 0.33398402259808585), (22950, 0.3531
6717190921404), (23297, 0.5004636069609789), (23437, 0.4987413913553328), (23534, 0.
3197278911564663), (23815, 0.4263208503741673), (24692, 0.5694947974515004), (25090,
0.3948592766275858), (26516, 0.4090668299050558), (28755, 0.6038099156819615), (2921
8, 0.48445214165180345), (30651, 0.4366879904979146), (30771, -0.06882270783070976),
(32599, 0.3877551020408152), (32738, 0.2144491380871325), (33509, 0.442981194961458
6), (35189, 0.5767195767195746), (35900, 0.1677474884872304), (35964, 0.870132019561
8422), (36946, -0.22584245794304467), (42734, 0.5270462766947298), (44758, 0.3763158
70913241), (46237, 0.39140669201059003), (46750, 0.2548235957188127), (50738, 0.0741
6029942875749), (53735, -0.3785694365397866), (56061, 0.36809892920177234), (57117,
```

```
0.004329004329005308), (57474, 0.4534083005202972), (57604, 0.5122408325718828), (58
040, 0.5919964701792331), (61364, 0.4371169495059537), (61705, 0.5270462766947298),
(64563, 0.13950659293219853), (65856, 0.3575514737971108), (66364, 0.251629955979423
2), (67204, 0.4000850754915663), (68783, 0.6129547322537948), (69392, 0.761904761904
7612), (71921, 0.5873015873015875), (73908, 0.14693487196685937), (74551, 0.31694749
76507784), (74858, 0.18077538151554676), (75161, 0.5253423888557812), (77194, 0.2599
8142027431187), (78629, 0.4730496064105083), (78871, 0.5144957554275268), (80261, 0.
706349206349205), (80485, 0.4408247953269107), (80652, -0.3690476190476181), (81627,
0.5731174625567711), (83951, -0.20001902768532487), (84039, 0.5935619399945083), (84
513, 0.48051948051948096), (86367, 0.6962116413460882), (87690, 0.0232357160225841
2), (87926, 0.5703514195350368), (88303, 0.22220903826313188), (89432, -0.0768112297
2277425), (90018, 0.2682265995890754), (90278, 0.7254233709051506), (90613, 0.463179
2885072645), (90863, 0.7453559924999297), (94166, -0.250697911303091), (94559, 0.106
47942749998995), (95198, 0.5698409630896697), (96736, 0.46106944597707333), (98070,
0.035493142499996654), (99970, 0.46275866639854907), (100428, 0.4225959332265023),
(101361, 0.3583820585161462), (103571, 0.08314646378209262), (104481, 0.459817027866
9846), (105554, 0.6279547345515971), (107538, 0.16250003516708023), (109101, 0.47619
047619047605)])
```

In [474...

```
pearsonDF = pd.DataFrame.from_dict(pearsonCorrelationDict, orient='index')
pearsonDF.columns = ['similarityIndex']
pearsonDF['userId'] = pearsonDF.index
pearsonDF.index = range(len(pearsonDF))
pearsonDF.head()
```

Out[474...

	similarityIndex	userId
0	-0.025566	815
1	0.289016	1502
2	0.547619	1599
3	0.498741	1625
4	0.007437	1950

The top 50 similar users to input user

Now that we have all of these pearson correlations calculated and place into a nice-neat dataframe, we should sort the users by their similarity.

In [578...

```
topUsers=pearsonDF.sort_values(by='similarityIndex', ascending=False)[0:70]
topUsers.head()
```

Out[578...

	similarityIndex	userId
42	0.870132	35964
10	0.810267	7235
5	0.774892	3186
62	0.761905	69392
16	0.759963	12921

But we should also check, how unrelated is the "bottom" of the 30 users?

In [579…
```
topUsers.tail()
```

Out[579…

	similarityIndex	userId
1	0.289016	1502
22	0.280906	17757
83	0.268227	90018
68	0.259981	77194
47	0.254824	46750

30 appears to have been a reasonable number of users to pick. I initially started with 50 users but the bottom correlation was reaching as low as 30% correlated. The correlation is not negative and are all moderately positive(similar preferences) at a limit of 30 users but we can use a weighted rating system to get a better subset of recommendations/more similar tastest still. If we were to limit our recommendations based off top 25 users, our simularity index (pearson correlation) would increase to 0.526 as the minimum. If we include all 100 nearest neighbours, we get into negative correlation and 70 neighbours puts us at a minimum correlation of 0.25

Rating of selected users to all movies

We're going to do this by taking the weighted average of the ratings of the movies using the Pearson Correlation as the weight. But to do this, we first need to get the movies watched by the users in our *pearsonDF* from the ratings dataframe and then store their correlation in a new column called _similarityIndex". This is achieved below by merging of these two tables.

In [580…
```
topUsersRating=topUsers.merge(ratings_df, left_on='userId', right_on='userId', how='
topUsersRating.head()
```

Out[580…

	similarityIndex	userId	movieId	rating
0	0.870132	35964	1	3.0
1	0.870132	35964	2	1.5
2	0.870132	35964	6	4.0
3	0.870132	35964	10	3.0
4	0.870132	35964	15	2.0

Now all we need to do is simply multiply the movie rating by its weight (the similarity index), then sum up the new ratings and divide it by the sum of the weights.

We can easily do this by simply multiplying two columns, then grouping up the dataframe by movieId and then dividing two columns:

It shows the idea of all similar users to candidate movies for the input user:

In [581…
```
#Multiplies the similarity by the user's ratings
topUsersRating['weightedRating'] = topUsersRating['similarityIndex']*topUsersRating[
topUsersRating.head()
```

Out[581…

	similarityIndex	userId	movieId	rating	weightedRating
0	0.870132	35964	1	3.0	2.610396
1	0.870132	35964	2	1.5	1.305198
2	0.870132	35964	6	4.0	3.480528
3	0.870132	35964	10	3.0	2.610396
4	0.870132	35964	15	2.0	1.740264

In [582…
```
#Applies a sum to the topUsers after grouping it up by userId
tempTopUsersRating = topUsersRating.groupby('movieId').sum()[['similarityIndex','wei
tempTopUsersRating.columns = ['sum_similarityIndex','sum_weightedRating']
tempTopUsersRating.head()
```

Out[582…

movieId	sum_similarityIndex	sum_weightedRating
1	34.898724	126.436886
2	34.898724	91.703715
3	15.857913	38.770038
4	2.599188	4.564155
5	13.632931	34.444658

In [583…
```
#Creates an empty dataframe
recommendation_df = pd.DataFrame()
#Now we take the weighted average
recommendation_df['weighted average recommendation score'] = tempTopUsersRating['sum
recommendation_df['movieId'] = tempTopUsersRating.index
recommendation_df.head()
```

Out[583…

movieId	weighted average recommendation score	movieId
1	3.622966	1
2	2.627710	2
3	2.444839	3

movieId	weighted average recommendation score	movieId
4	1.755992	4
5	2.526578	5

Now let's sort it and see the top 20 movies that the algorithm recommended!

In [584…
```
recommendation_df = recommendation_df.sort_values(by='weighted average recommendatic
recommendation_df.head(10)
```

Out[584…

movieId	weighted average recommendation score	movieId
129401	5.0	129401
4769	5.0	4769
117109	5.0	117109
8423	5.0	8423
88203	5.0	88203
63237	5.0	63237
935	5.0	935
6921	5.0	6921
8582	5.0	8582
134583	5.0	134583

If we look at our recommender system results, they are not stellar. They contain a moderate amount of older films perhaps users would not enjoy as much as newer films and foreign films that may not be available in their native language or might not be their preference.

In [585…
```
movies_df.loc[movies_df['movieId'].isin(recommendation_df.head(30)['movieId'].tolist
```

Out[585…

	movieId	title	year
622	629	Rude	1995
661	670	World of Apu, The (Apur Sansar)	1959
918	935	Band Wagon, The	1953
4674	4769	Into the Arms of Strangers: Stories of the Kin...	2000
4711	4806	Shop on Main Street, The (Obchod na korze)	1965
6787	6896	Shoah	1985
6811	6921	Man of Marble (Czlowiek z Marmuru)	1977

	movieId	title	year
6963	7074	Navigator, The	1924
7810	8423	Kid Brother, The	1927
7901	8582	Manufacturing Consent: Noam Chomsky and the Media	1992
8067	8749	Beautiful Troublemaker, The (La belle noiseuse)	1991
8355	25753	Greed	1924
8361	25763	Pandora's Box (Büchse der Pandora, Die)	1929
8375	25782	Boudu Saved From Drowning (Boudu sauvé des eaux)	1932
8510	25952	Letter to Three Wives, A	1949
11947	53774	Mystery of Picasso, The (Le mystère Picasso)	1956
12338	56779	I Don't Want to Sleep Alone (Hei yan quan)	2006
12904	60904	Heart of a Dog (Sobachye serdtse)	1988
13135	63237	Happily Ever After	1993
13515	66744	Divo, Il	2008
14737	73501	Pekka ja Pätkä Suezilla	1958
17284	87079	Trust	2010
17558	88203	Car Bonus (Autobonus)	2001
19381	96062	Jekyll	2007
24560	115467	Harmontown	2014
25046	117109	Too Many Cooks	2014
27940	129401	Kevin Smith: Sold Out - A Threevening with Kev...	2008
29398	134583	Misery Loves Comedy	2015
30440	138104	Justice League: Gods and Monsters	2015
31094	140359	Doctor Who: The Waters of Mars	2009

Some people dislike subtitles and audio-dub overs. Your boss wanted to know just how many people dislike foreign films, so he hired a polling company. Let's pretend "market research" informed us, 98% or more of users polled responded unfavorably to the question "Do you like to watch movies older than 1980?" and "Do you like to watch movies older 95% to older than 2000?". As a result, your boss says the recommender you make shouldn't include movies older than "ehh, let's say 98 because I like that year. It was the year my daughter was born and no foreign stuff either!"

In [586…

```
first_recommendation_stage_df = movies_df.loc[movies_df['movieId'].isin(recommendati
#Let's drop years < 1998
first_recommendation_stage_df['year'] = first_recommendation_stage_df['year'].apply(
first_recommendation_stage_df.drop(first_recommendation_stage_df[first_recommendatic
#Let's drop the foreign films. Note each foreign film tends to have a '(alternative
#The str.find() function will return -1 when a string is not-found. So we are negati
```

```
#We will now drop these indexes from our recommender system.
first_recommendation_stage_df.drop(first_recommendation_stage_df[first_recommendatic
#One can also achieve this with an alternative algorithm to capture, foreign titled
#The downside is while it is more precise than removing all ( possibly, it would req
first_recommendation_stage_df.drop(first_recommendation_stage_df[first_recommendatic
first_recommendation_stage_df.drop(first_recommendation_stage_df[first_recommendatic
first_recommendation_stage_df.drop(first_recommendation_stage_df[first_recommendatic
#etc. Ideally, use a "for" loop and a list of foreign-accent characters. for c in [ë
```

```
C:\Users\WBurc\AppData\Local\Temp\ipykernel_25144\3007877730.py:3: SettingWithCopyWa
rning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/u
ser_guide/indexing.html#returning-a-view-versus-a-copy
  first_recommendation_stage_df['year'] = first_recommendation_stage_df['year'].appl
y(pd.to_numeric) #We'll get an error if we try to 'compare' str to int so we have to
convert the year column from str in csv to int
C:\Users\WBurc\AppData\Local\Temp\ipykernel_25144\3007877730.py:4: SettingWithCopyWa
rning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/u
ser_guide/indexing.html#returning-a-view-versus-a-copy
  first_recommendation_stage_df.drop(first_recommendation_stage_df[first_recommendat
ion_stage_df.year < 1998].index,inplace=True) #we are dropping index(row numbers) wh
ere year < 1998 and doing inplace so df is updated without an intermediate variable
needed
C:\Users\WBurc\AppData\Local\Temp\ipykernel_25144\3007877730.py:8: SettingWithCopyWa
rning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/u
ser_guide/indexing.html#returning-a-view-versus-a-copy
  first_recommendation_stage_df.drop(first_recommendation_stage_df[first_recommendat
ion_stage_df['title'].apply(lambda x: x.find('(')) != -1].index,inplace=True)
C:\Users\WBurc\AppData\Local\Temp\ipykernel_25144\3007877730.py:11: SettingWithCopyW
arning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/u
ser_guide/indexing.html#returning-a-view-versus-a-copy
  first_recommendation_stage_df.drop(first_recommendation_stage_df[first_recommendat
ion_stage_df['title'].apply(lambda x: x.find('ë')) != -1].index,inplace=True)
C:\Users\WBurc\AppData\Local\Temp\ipykernel_25144\3007877730.py:12: SettingWithCopyW
arning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/u
ser_guide/indexing.html#returning-a-view-versus-a-copy
  first_recommendation_stage_df.drop(first_recommendation_stage_df[first_recommendat
ion_stage_df['title'].apply(lambda x: x.find('ö')) != -1].index,inplace=True)
C:\Users\WBurc\AppData\Local\Temp\ipykernel_25144\3007877730.py:13: SettingWithCopyW
arning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/u
ser_guide/indexing.html#returning-a-view-versus-a-copy
```

```
  first_recommendation_stage_df.drop(first_recommendation_stage_df[first_recommendat
ion_stage_df['title'].apply(lambda x: x.find('è')) != -1].index,inplace=True)
```

In [587…

```
first_recommendation_stage_df.loc[first_recommendation_stage_df['movieId'].isin(recc
```

Out[587…

	movieId	title	year
4674	4769	Into the Arms of Strangers: Stories of the Kin...	2000.0
9165	27002	From the Earth to the Moon	1998.0
10483	38499	Angels in America	2003.0
10849	43832	Call of Cthulhu, The	2005.0
12110	55063	My Winnipeg	2007.0
12803	60295	Up the Yangtze	2007.0
13515	66744	Divo, Il	2008.0
15153	77154	Waking Sleeping Beauty	2009.0
15535	79006	Empire of Dreams: The Story of the 'Star Wars'...	2004.0
16874	85190	Public Speaking	2010.0
17284	87079	Trust	2010.0
19381	96062	Jekyll	2007.0
20928	101971	Never Sleep Again: The Elm Street Legacy	2010.0
21383	103602	Craig Ferguson: I'm Here To Help	2013.0
22562	107718	State of Play	2003.0
22613	107910	I Know That Voice	2013.0
22805	108660	Deceptive Practice: The Mysteries and Mentors ...	2012.0
24560	115467	Harmontown	2014.0
25046	117109	Too Many Cooks	2014.0
25904	120811	Patton Oswalt: Finest Hour	2011.0
25905	120813	Patton Oswalt: My Weakness Is Strong	2009.0
25906	120815	Patton Oswalt: Werewolves and Lollipops	2007.0
26697	124131	Lewis Black: Stark Raving Black	2009.0
26723	124273	Kevin Smith: Too Fat For 40	2010.0
27602	128091	Craig Ferguson: A Wee Bit o' Revolution	2009.0
27654	128320	Monty Python: Almost the Truth - Lawyers Cut	2009.0
27662	128366	Patton Oswalt: Tragedy Plus Comedy Equals Time	2014.0
27788	128812	Drew: The Man Behind the Poster	2013.0
27940	129401	Kevin Smith: Sold Out - A Threevening with Kev...	2008.0
28536	131724	The Jinx: The Life and Deaths of Robert Durst	2015.0

	movieId	title	year
29398	134583	Misery Loves Comedy	2015.0
30440	138104	Justice League: Gods and Monsters	2015.0
31094	140359	Doctor Who: The Waters of Mars	2009.0
33187	147372	Doctor Who: Last Christmas	2014.0
33188	147374	Doctor Who: The Doctor, the Widow and the Ward...	2011.0
33189	147376	Doctor Who: A Christmas Carol	2010.0
33190	147378	Doctor Who: Planet of the Dead	2009.0
33192	147382	Doctor Who: Voyage Of The Damned	2007.0
33193	147384	Doctor Who: The Runaway Bride	2007.0
34054	150856	Making a Murderer	2015.0

After examining the output of our Pearson correlation based collaborate filter versus our item-item based filter which examines the content/genres of the films, the item-item based filter performed much better at making recommendations in my opinion. If we were to do (A/B) testing, I think "B" or the other recommender would be the superior choice. More sample users/data would need to be ran through to confirm. If one wanted to improve this project, one should consoldate the steps into a single function that runs each step so you can simple go recommend_movies_to(user_ID) where the userID is looked up, data collected for past preferences/ratings/movies they rated and it outputs the recommendation as a list. A list we can return/make available as an API to our GUI designer/front-end developer for our movie-watching/streaming site.

Advantages and Disadvantages of Collaborative Filtering

Advantages

- Takes other user's ratings into consideration
- Doesn't need to study or extract information from the recommended item
- Adapts to the user's interests which might change over time

Disadvantages

- Approximation function can be slow
- There might be a low amount of users to approximate
- Privacy issues when trying to learn the user's preferences