Published in The Startup

Bryan Tan Follow
Apr 19, 2020 · 15 min read · · Listen

How to Build Simple Recommender Systems in Python

User-based Collaborative Filtering using the Pearson Correlation Coefficient

Photo by Mollie Sivaram on Unsplash

1. Introduction

1.1 Background

Even though people's tastes may vary, they generally follow patterns. By that, I mean that there are similarities in the things that people tend to like. Or another way to look at it is that people tend to like things in the same category or things that share the same characteristics. For example, if you've recently purchased a book on "Machine Learning in Python" and you've enjoyed reading it, it's very likely that you'll also enjoy reading a book on Data Visualization. People also tend to have similar tastes to those of the people they're close to in their lives. Recommender systems try to capture these patterns and similar behaviours, to help predict what else you might like.

Recommender systems have many applications that I'm sure you're already familiar with. Indeed, Recommender systems are usually at play on many websites. For example, suggesting books on *Amazon* and movies on *Netflix*. In fact, everything on *Netflix's* website is driven by customer selection. If a certain movie gets viewed frequently enough, *Netflix's* recommender system ensures that that movie gets an increasing number of recommendations. Another example can be found in a daily-use mobile app, where a recommender engine is used to recommend anything from where to eat or what job to apply to. On social media, sites like *Facebook* or *LinkedIn*, regularly recommend friendships.

Most streaming services utilize recommender systems.

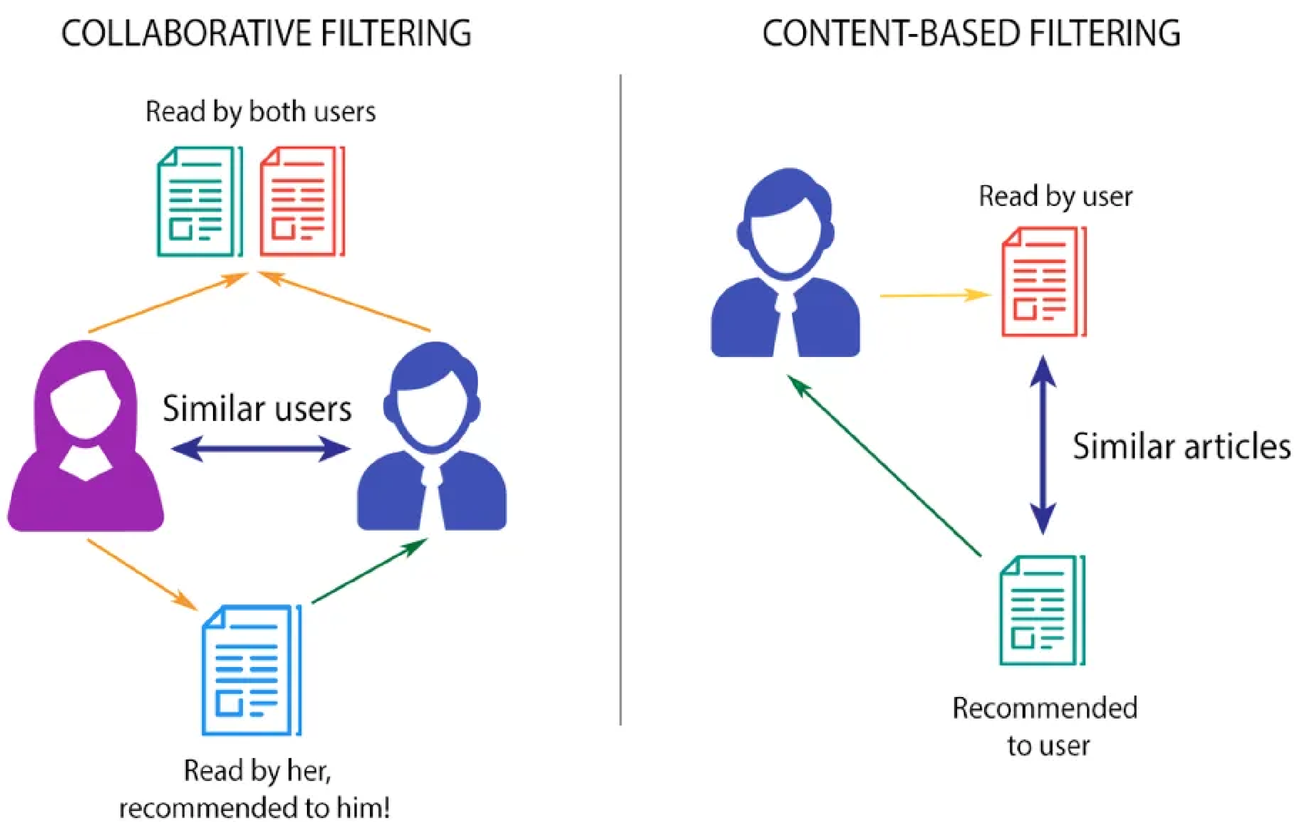
Recommender systems are even used to personalize your experience on the web. For example, when you go to a news platform website, a recommender system will make note of the types of stories that you clicked on and make recommendations on which types of stories you might be interested in reading in the future. There are many of these types of examples and they are growing in number every day.

One of the main advantages of using recommendation systems is that users get broader exposure to many different products they might be interested in. This exposure encourages users towards the continual usage or purchase of their products. Not only does this provide a better experience for the user but it benefits the service provider, as well, with increased potential revenue and better security for its customers.

1.2 Types of Recommender Systems

There are generally 2 main types of recommendation systems: ***Content-based*** and ***collaborative filtering***. The main difference between each can be summed up by the type of statement that a consumer might make. For instance, the main paradigm of a content-based recommendation system is driven by the statement: "*Show me more of the same of what I've liked before.*"

Content-based systems try to figure out what a user's favorite aspects of an item are, and then make recommendations on items that share those aspects. Collaborative filtering is based on a user saying, "*Tell me what's popular among my neighbours because I might like it too.*" Collaborative filtering techniques find similar groups of users and provide recommendations based on similar tastes within that group. In short, it assumes that a user might be interested in what similar users are interested in. There are also hybrid recommender systems that combine various mechanisms. Though the focal point of this article is on the collaborative filtering approach.



Collaborative Filtering vs Content-Based Filtering

1.3 Implementation of Recommender Systems

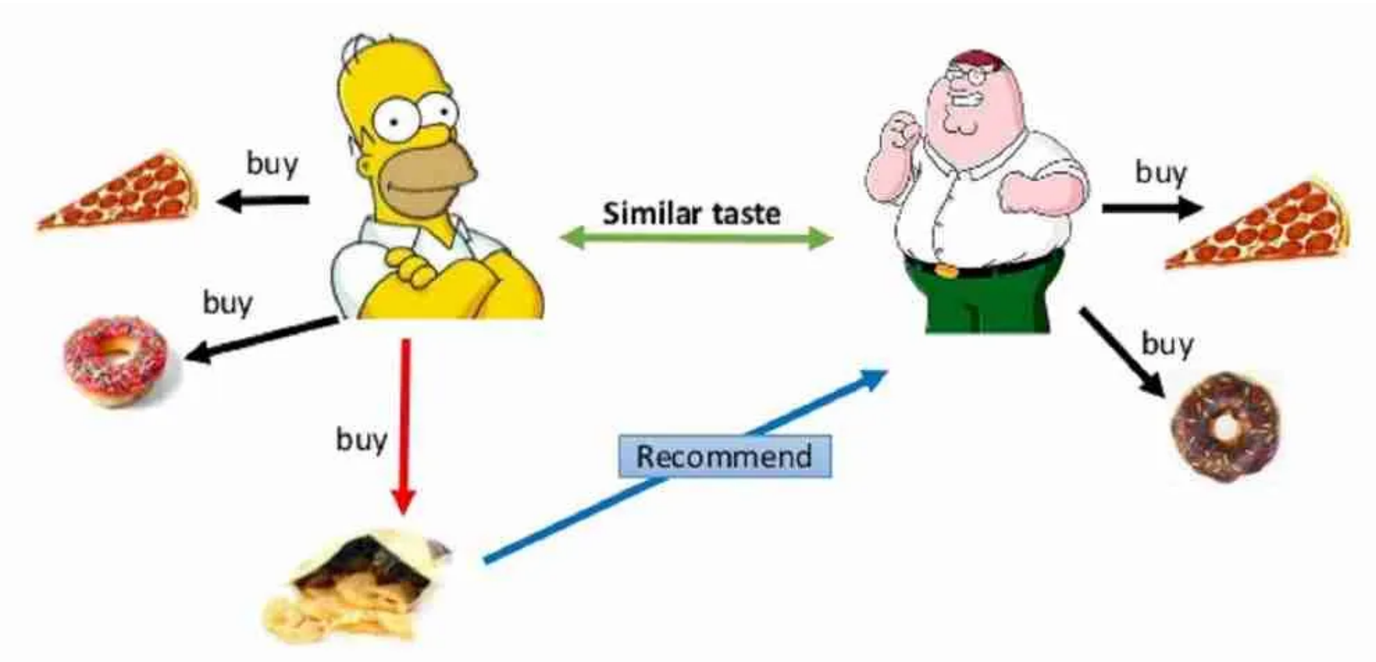
In terms of implementing recommender systems, there are 2 types: ***memory-based*** and ***model-based***. In memory-based approaches, we use the entire user-item dataset to generate a recommendation system. It uses statistical techniques to approximate users or items. Examples of these techniques include ***Pearson Correlation***, ***Cosine Similarity, Euclidean Distance***, and among others. In model-based approaches, a model of users is developed in an attempt to learn their preferences. Models can be created using machine learning techniques like regression, clustering, classification, and so on.

2. Collaborative Filtering

2.1 Intuition

Collaborative filtering is based on the fact that relationships exist between products and people's interests. Many recommendation systems use collaborative filtering to find these relationships and to give an accurate recommendation of a product that the user might like or be interested in. Collaborative filtering has basically two approaches: ***user-based*** and ***item-based***. User-based collaborative filtering is based on the user similarity or neighborhood. Item-based collaborative filtering is based on similarity among items. Let's first understand the intuition behind the user-based approach.

In user-based collaborative filtering, we have an active user for whom the recommendation is aimed at. The collaborative filtering engine first looks for users who are similar to that particular active user, that is, users who share the active user's rating patterns. Collaborative filtering bases this similarity on things like history, preference, and choices that users make when buying, watching, or enjoying something, for example, movies that similar users have rated highly. Then it uses the ratings from these similar users to predict the possible ratings by the active user for a movie that they had not previously watched. For instance, if two users are similar or, are neighbors in terms of their interested movies, we can recommend a movie to the active user that their neighbor has already seen.



User-based Collaborative Filtering

2.2 Algorithm

Now, let's dive into the algorithm to see how all of this works.

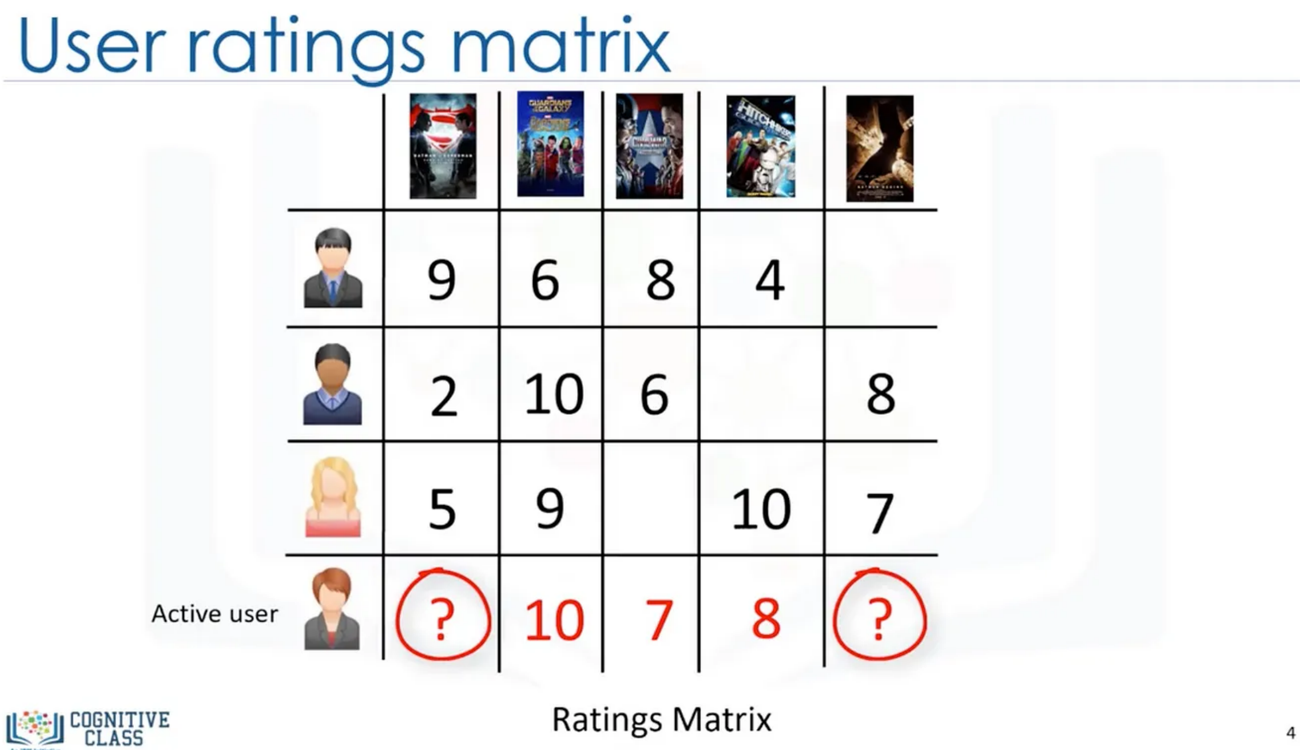


Fig 1

Assume that we have a simple ***user-item matrix,*** which shows the ratings of four users for five different movies. Let's also assume that our active user has watched and rated three out of these five movies. Let's find out which of the two movies that our active user hasn't watched should be recommended to her.

The first step is to discover how similar the active user is to the other users. How do we do this? Well, this can be done through several different statistical and vectorial techniques such as distance or similarity measurements including ***Euclidean Distance***, ***Pearson Correlation***, ***Cosine Similarity***, and so on. To calculate the level of similarity between two users, we use the three movies that both the users have rated in the past.

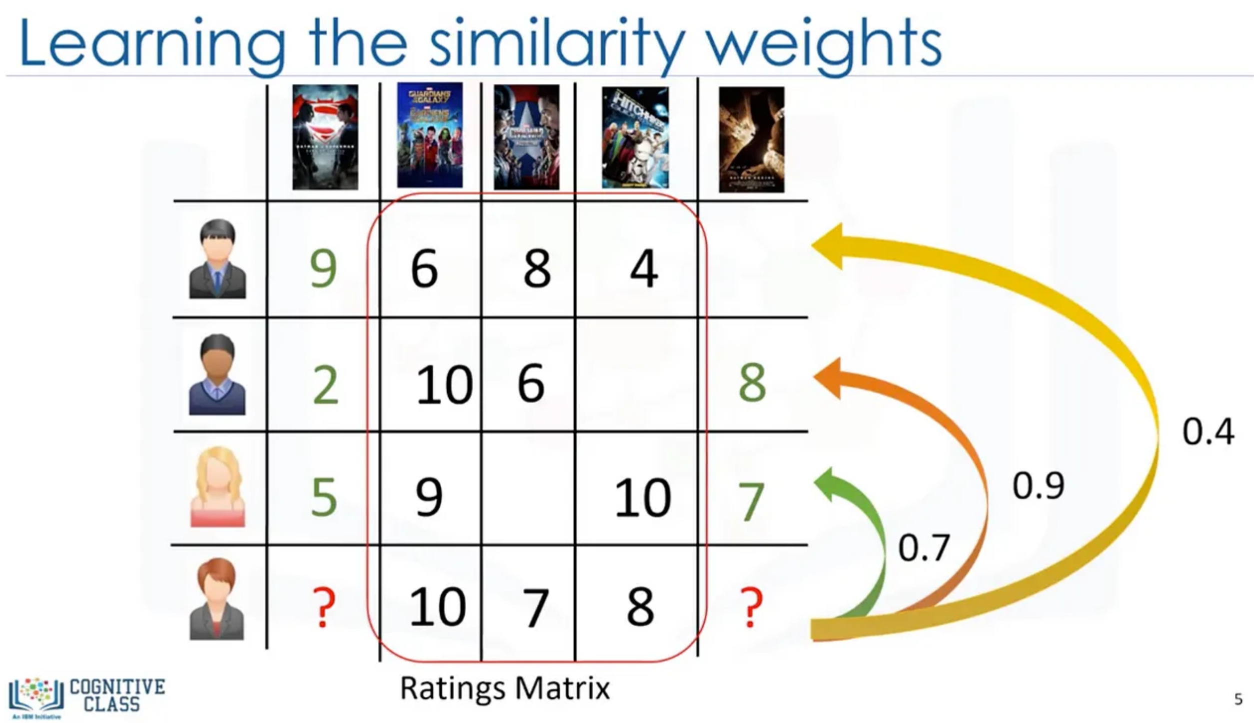


Fig 2

Regardless of what we use for similarity measurement, for example, the similarity could be *0.7, 0.9,* and *0.4* between the active user and other users. These numbers represent similarity weights or proximity of the active user to other users in the dataset. The next step is to create a ***weighted rating matrix***. We just calculated the similarity of users to our active user in *Fig 2; n*ow, we can use it to calculate the possible opinion of the active user about our two target movies. This is achieved by multiplying the similarity weights to the user ratings.

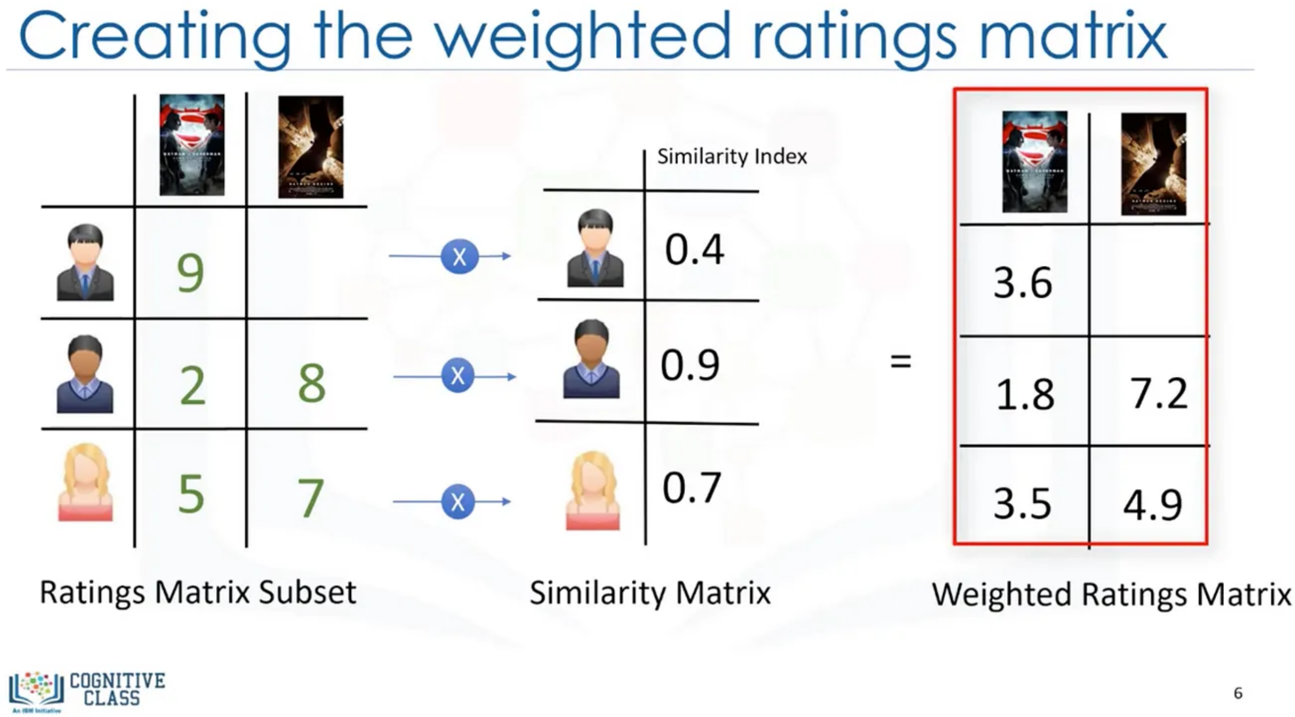


Fig 3

It results in a ***weighted ratings matrix,*** which represents the user's neighbours' opinions about our two candidate movies for recommendation. In fact, it incorporates the behaviour of other users and gives more weight to the ratings of those users who are more similar to the active user.

Now, we can generate the recommendation matrix by aggregating all of the weighted rates. However, as three users rated the first potential movie and two users rated the second movie, we have to ***normalize*** the weighted rating values. We do this by dividing the ***sum of weighted ratings*** by the ***sum of the similarity index*** for users.

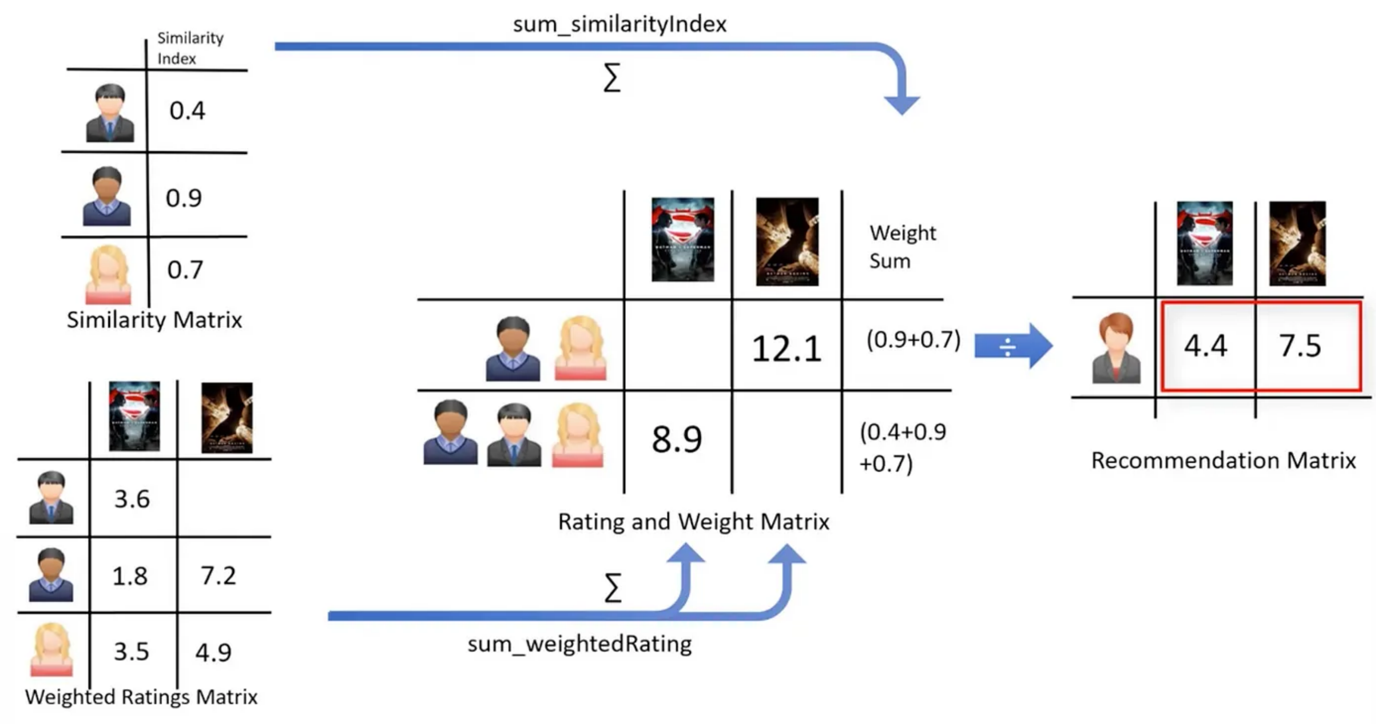


Fig 4

The result is the potential rating that our active user will give to these movies based on her similarity to other users. It is obvious that we can use it to rank the movies for providing recommendation to our active user.

Now, let's examine the difference between user-based and item-based collaborative filtering.

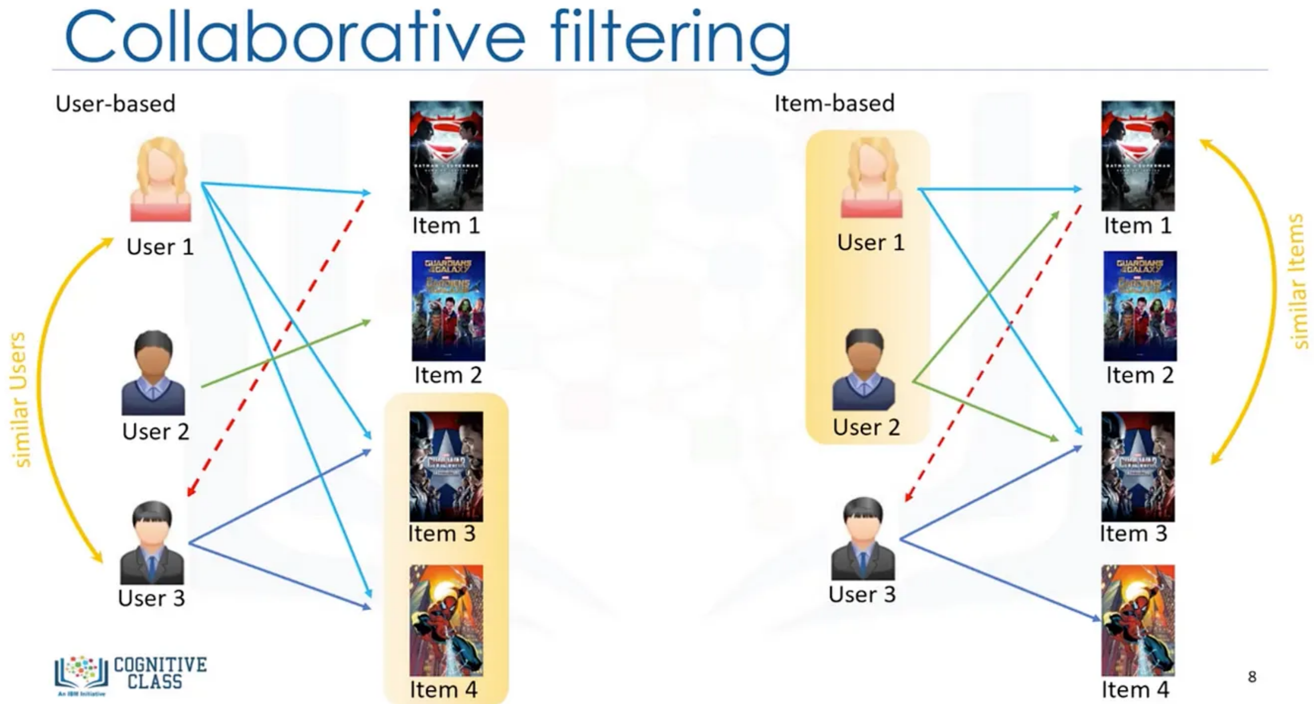


Fig 5

In the user-based approach, the recommendation is based on users of the same neighbourhood with whom he or she shares common preferences. For example, as *User 1* and *User 3* both liked *Item 3* and *Item 4*, we consider them as similar — or neighbour users — and recommend *Item 1* which is positively rated by *User 1* to *User 3*.

In the item-based approach, similar items build neighbourhoods on the behaviour of users (not based on their contents!). For example, *Item 1* and *Item 3* are considered neighbours as they were positively rated by both *User 1* and *User 2*. So, *Item 1* can be recommended to *User 3* as he or she has already shown interest in *Item 3*. Therefore, the recommendations here are based on the items in the neighborhood that a user might prefer.

2.3 Challenges of Collaborative Filtering

Collaborative filtering is a very effective recommendation system. However, there are some challenges with it as well. One of them is ***data sparsity***. Data sparsity happens when you have a large data set of users who generally rate only a limited number of items. As mentioned, collaborative based recommenders can only predict the scoring of an item if there are other existing users who have rated it. Due to sparsity, we might not have enough ratings in the user-item dataset which makes it impossible to provide proper recommendations.

Another issue to keep in mind is something called ***cold start***. Cold start refers to the difficulty the recommendation system has when there is a new user, and as such, a profile doesn't exist for them yet. Cold start can also happen when we have a new item which has not received a rating.

Scalability can become an issue as well. As the number of users or items increases, and the amount of data expands, collaborative filtering algorithms will begin to suffer performance dips, simply due to growth and the similarity computation. There are some solutions for each of these challenges, such as using hybrid based recommender systems, but they are out of the scope of this topic.

3. Building a Simple Recommender System in Python

3.1 Acquiring Data

The dataset for this project was acquired from GroupLens.

3.2 Preprocessing

First, let's get all of the imports out of the way:

```
#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
```

Now let's read each file into their Dataframes:

```
#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')
```

Let's also take a peek at how each of them are organized:

	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

movies_df.head()

So each movie has a unique ID, a title with its release year along with it (which may contain unicode characters), and several different genres in the same field. Let's remove the `year` from the `title` column and store it a new `year` column by using the handy `extract` function that Pandas has.

```
#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 their titles

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 have appeared

movies_df['title'] = movies_df['title'].apply(lambda x: x.strip())
```

Let's look at the result:

	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

movies_df.head()

With that, let's also drop the genres column since we won't need it for this particular recommendation system:

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

Here's the final movies dataframe:

	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

movies_df.head()

Next, let's look at the ratings dataframe:

	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

ratings_df.head()

3.3 Collaborative Filtering

Now, time to start our work on the recommendation system.

The technique we're going to take a look at, as titled, is **Collaborative Filtering**, which is also known as **User-User Filtering**. As hinted by its alternate name, this technique uses other users to recommend items to the input user. It attempts to find users that have similar preferences and opinions as the input and then recommends items that they have liked to the input. There are several methods of finding similar users (even some making use of Machine Learning). The one we will be using here is going to be based on the **Pearson Correlation Function**.

To recap the process for creating a user-based recommendation system:

- Select a user with the movies the user has watched
- Based on his rating to movies, find the top X neighbours
- Get the watched movie record of the user for each neighbour.
- Calculate a similarity score using some formula
- Recommend the items with the highest score

Let’s begin by creating an input user to recommend movies to:

Notice: To add more movies, simply increase the amount of elements in the userInput. Feel free to add more in! Just be sure to write it in with capital letters and if a movie starts with a “The”, like “The Matrix” then write it in like this: ‘Matrix, The’.

```
userInput = [
{‘title’:’Breakfast Club, The’, ‘rating’:5},
{‘title’:’Toy Story’, ‘rating’:3.5},
{‘title’:’Jumanji’, ‘rating’:2},
{‘title’:”Pulp Fiction”, ‘rating’:5},
{‘title’:’Akira’, ‘rating’:4.5}
]
inputMovies = pd.DataFrame(userInput)
inputMovies
```

	title	rating
0	Breakfast Club, The	5.0
1	Toy Story	3.5
2	Jumanji	2.0
3	Pulp Fiction	5.0
4	Akira	4.5

inputMovies

With the input complete, let’s extract the input movies’ ID’s from the movies dataframe and add them into it.

We can achieve this by first filtering out the rows that contain the input movies' `title` and then merging this subset with the input dataframe. We also drop unnecessary columns for the input to save memory space.

```
#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
```

	movieId	title	rating
0	1	Toy Story	3.5
1	2	Jumanji	2.0
2	296	Pulp Fiction	5.0
3	1274	Akira	4.5
4	1968	Breakfast Club, The	5.0

inputMovies

Now with the `movieId` in our input, we can now get the subset of users that have watched and reviewed the movies in our input.

```
#Filtering out users that have watched movies that the input has watched and storing it

userSubset = ratings_df[ratings_df['movieId'].isin(inputMovies['movieId'].tolist())]

userSubset.head()
```

	userId	movieId	rating
19	4	296	4.0
441	12	1968	3.0
479	13	2	2.0
531	13	1274	5.0
681	14	296	2.0

userSubset.head()

We now group up the rows by `userId`:

```
#Groupby creates several sub dataframes where they all have the same value in the column specified as the parameter

userSubsetGroup = userSubset.groupby(['userId'])
```

lets look at one of the users, e.g. the one with `userID` =1130

	userId	movieId	rating
104167	1130	1	0.5
104168	1130	2	4.0
104214	1130	296	4.0
104363	1130	1274	4.5
104443	1130	1968	4.5

userSubsetGroup.get_group(1130)

Let’s also sort these groups so the users that share the most movies in common with the input have higher priority. This provides a richer recommendation since we won’t go through every single user.

```
#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 lets look at the first user:

```
[(75,       userId  movieId  rating
   7507       75        1     5.0
   7508       75        2     3.5
   7540       75      296     5.0
   7633       75     1274     4.5
   7673       75     1968     5.0), (106,       userId  movieId  rating
   9083      106        1     2.5
   9084      106        2     3.0
   9115      106      296     3.5
   9198      106     1274     3.0
   9238      106     1968     3.5), (686,        userId  movieId  rating
   61336      686        1     4.0
   61337      686        2     3.0
   61377      686      296     4.0
   61478      686     1274     4.0
   61569      686     1968     5.0)]
```

userSubsetGroup[0:3]

3.3.1 Similarity of users to input user

Next, we are going to compare all users (not really all !!!) to our specified user and find the one that is most similar.
we're going to find out how similar each user is to the input through the ***Pearson Correlation Coefficient***. It is used to measure the strength of a linear association between two variables. The formula for finding this coefficient between sets X and Y with N values can be seen in the image below:

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

Pearson Correlation Coefficient formula

Why Pearson Correlation?

Pearson correlation is invariant to scaling, i.e. multiplying 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). This is a pretty important property in recommendation systems because for example two users might rate two series of items totally different in terms of absolute rates, but they would be similar users (i.e. with similar ideas) with similar rates in various scales .

The values given by the formula vary from r = -1 to r = 1, where 1 forms a direct correlation between the two entities (it means a perfect positive correlation) and -1 forms a perfect negative correlation.

In our case, a 1 means that the two users have similar tastes while a -1 means the opposite.

We will select a subset of users to iterate through. This limit is imposed because we don't want to waste too much time going through every single user.

```
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 `userId` and the value is the coefficient.

```
#Store the Pearson Correlation in a dictionary, where the key is the user Id and the value is the coefficient

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 mixed up later on

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 facilitate future calculations
```

```
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 y

Sxx = sum([i**2 for i in tempRatingList]) —
pow(sum(tempRatingList),2)/float(nRatings)

Syy = sum([i**2 for i in tempGroupList]) —
pow(sum(tempGroupList),2)/float(nRatings)

Sxy = sum( i*j for i, j in zip(tempRatingList, tempGroupList)) —
sum(tempRatingList)*sum(tempGroupList)/float(nRatings)

#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
```

Converting the dictionary to a dataframe:

```
pearsonDF = pd.DataFrame.from_dict(pearsonCorrelationDict,
orient=’index’)

pearsonDF.columns = [‘similarityIndex’]

pearsonDF[‘userId’] = pearsonDF.index

pearsonDF.index = range(len(pearsonDF))

pearsonDF.head()
```

	similarityIndex	userId
0	0.827278	75
1	0.586009	106
2	0.832050	686
3	0.576557	815
4	0.943456	1040

pearsonDF.head()

3.3.2 The top x similar users to input user

Now let's get the top 50 users that are most similar to the input:

```
topUsers=pearsonDF.sort_values(by='similarityIndex', ascending=False)
[0:50]

topUsers.head()
```

	similarityIndex	userId
64	0.961678	12325
34	0.961538	6207
55	0.961538	10707
67	0.960769	13053
4	0.943456	1040

topUsers.head()

Now, let's start recommending movies to the input user.

3.3.3 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.

```
topUsersRating=topUsers.merge(ratings_df, left_on='userId', right_on='userId', how='inner')

topUsersRating.head()
```

	similarityIndex	userId	movieId	rating
0	0.961678	12325	1	3.5
1	0.961678	12325	2	1.5
2	0.961678	12325	3	3.0
3	0.961678	12325	5	0.5
4	0.961678	12325	6	2.5

topUsersRating.head()

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:

```
#Multiplies the similarity by the user’s ratings

topUsersRating[‘weightedRating’] =
topUsersRating[‘similarityIndex’]*topUsersRating[‘rating’]

topUsersRating.head()
```

	similarityIndex	userId	movieId	rating	weightedRating
0	0.961678	12325	1	3.5	3.365874
1	0.961678	12325	2	1.5	1.442517
2	0.961678	12325	3	3.0	2.885035
3	0.961678	12325	5	0.5	0.480839
4	0.961678	12325	6	2.5	2.404196

topUsersRating.head()

```
#Applies a sum to the topUsers after grouping it up by userId

tempTopUsersRating = topUsersRating.groupby('movieId').sum()
[['similarityIndex','weightedRating']]

tempTopUsersRating.columns =
['sum_similarityIndex','sum_weightedRating']

tempTopUsersRating.head()
```

	sum_similarityIndex	sum_weightedRating
movieId		
1	38.376281	140.800834
2	38.376281	96.656745
3	10.253981	27.254477
4	0.929294	2.787882
5	11.723262	27.151751

tempTopUsersRating.head()

```
#Creates an empty dataframe

recommendation_df = pd.DataFrame()

#Now we take the weighted average

recommendation_df['weighted average recommendation score'] =
tempTopUsersRating['sum_weightedRating']/tempTopUsersRating['sum_simil
arityIndex']

recommendation_df['movieId'] = tempTopUsersRating.index

recommendation_df.head()
```

movieId	weighted average recommendation score	movieId
1	3.668955	1
2	2.518658	2
3	2.657941	3
4	3.000000	4
5	2.316058	5

recommendation_df.head()

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

```
recommendation_df = recommendation_df.sort_values(by='weighted average
recommendation score', ascending=False)

recommendation_df.head(10)
```

movieId	weighted average recommendation score	movieId
5073	5.0	5073
3329	5.0	3329
2284	5.0	2284
26801	5.0	26801
6776	5.0	6776
6672	5.0	6672
3759	5.0	3759
3769	5.0	3769
3775	5.0	3775
90531	5.0	90531

recommendation_df.head()

	movieId	title	year
2200	2284	Bandit Queen	1994
3243	3329	Year My Voice Broke, The	1987
3669	3759	Fun and Fancy Free	1947
3679	3769	Thunderbolt and Lightfoot	1974
3685	3775	Make Mine Music	1946
4978	5073	Son's Room, The (Stanza del figlio, La)	2001
6563	6672	War Photographer	2001
6667	6776	Lagaan: Once Upon a Time in India	2001
9064	26801	Dragon Inn (Sun lung moon hak chan)	1992
18106	90531	Shame	2011

movies_df.loc[movies_df['movieId'].isin(recommendation_df.head(10)['movieId'].tolist())]

4. 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 of amount of users to approximate
- Privacy issues when trying to learn the user's preferences

References

[1] Github repo for notebook:

TheClub4/collaborative_filtering

Building a simple recommender system with collaborative filtering / user-user filtering approach. ...

github.com

[2] **Algorithm** *figures* courtesy of IBM Cognitive Class.

Blog - Cognitive Class

Free courses on Data Science, Artificial Intelligence, Machine Learning, Big Data, Blockchain, IoT, Cloud Computing and...

cognitiveclass.ai

Recommendation System Python Collaborative Filtering Netflix Data Science