Algorithms for Information Retrieval and Intelligence Web (UE20CS332) Assignment - 2: Analysis

Name	SRN
Saakshi H Srinivasan	PES2UG20CS290
Sakshi Hulageri	PES2UG20CS300
Samhitha R Nadig	PES2UG20CS304

Introduction

A movie recommendation system, or a movie recommender system, is an ML-based approach to filtering or predicting the users' film preferences based on their past choices and behavior. It's an advanced filtration mechanism that predicts the possible movie choices of the concerned user and their preferences towards a domain-specific item, aka movie.

The primary goal of movie recommendation systems is to filter and predict only those movies that a corresponding user is most likely to want to watch. The ML algorithms for these recommendation systems use the data about this user from the system's database. This data is used to predict the future behavior of the user concerned based on the information from the past.

Corpus

The data consists of 105339 ratings applied over 10329 movies.

The movies.csv dataset contains three columns:

- movieId: the ID of the movie
- title: movies title
- genres: movies genres

The ratings.csv dataset contains four columns:

- userId: the ID of the user who rated the movie.
- movieId: the ID of the movie
- ratings: ratings given by each user (from 0 to 5)
- Timestamp: The time the movie was rated.

Dataset link:
https://www.kaggle.com/code/ayushimishra2809/movie-recommendation-system/input

Notebook link:
https://colab.research.google.com/drive/1GKLoSZTVa-kLeGvNyfUkI61mLpxfB65f#scrollTo=jp5ZKinvRqnN

Exploratory Data Analysis (EDA)

```
import numpy as np
import pandas as pd
import os
import matplotlib.pyplot as plt
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import linear_kernel
from scipy import stats
import warnings
warnings.filterwarnings("ignore")
```

```
os.chdir('/Users/surykanthulageri/Desktop')
os.getcwd()
```

```
'/Users/surykanthulageri/Desktop'
```

```
movies=pd.read_csv('movies.csv')
ratings=pd.read_csv('ratings.csv')
```

```python
movies.head()
```

	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

```python
movies.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10329 entries, 0 to 10328
Data columns (total 3 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   movieId  10329 non-null  int64
 1   title    10329 non-null  object
 2   genres   10329 non-null  object
dtypes: int64(1), object(2)
memory usage: 242.2+ KB
```

```python
movies.shape
```

```
(10329, 3)
```

```python
movies.describe()
```

	movieId
count	10329.000000
mean	31924.282893
std	37734.741149
min	1.000000
25%	3240.000000
50%	7088.000000
75%	59900.000000
max	149532.000000

```
ratings.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 105339 entries, 0 to 105338
Data columns (total 4 columns):
 #   Column     Non-Null Count   Dtype  
---  ------     --------------   -----  
 0   userId     105339 non-null  int64  
 1   movieId    105339 non-null  int64  
 2   rating     105339 non-null  float64
 3   timestamp  105339 non-null  int64  
dtypes: float64(1), int64(3)
memory usage: 3.2 MB
```

```
ratings.shape
```

```
(105339, 4)
```

```
ratings.describe()
```

	userId	movieId	rating	timestamp
count	105339.000000	105339.000000	105339.000000	1.053390e+05
mean	364.924539	13381.312477	3.516850	1.130424e+09
std	197.486905	26170.456869	1.044872	1.802660e+08
min	1.000000	1.000000	0.500000	8.285650e+08
25%	192.000000	1073.000000	3.000000	9.711008e+08
50%	383.000000	2497.000000	3.500000	1.115154e+09
75%	557.000000	5991.000000	4.000000	1.275496e+09
max	668.000000	149532.000000	5.000000	1.452405e+09

From the above table we can conclude that:

- The average rating is 3.5 and minimum and maximum rating is 0.5 and 5 respectively.
- There are 668 users who have given their ratings for 149532 movies.

```python
df=pd.merge(ratings,movies, how='left',on='movieId')
df.head()
```

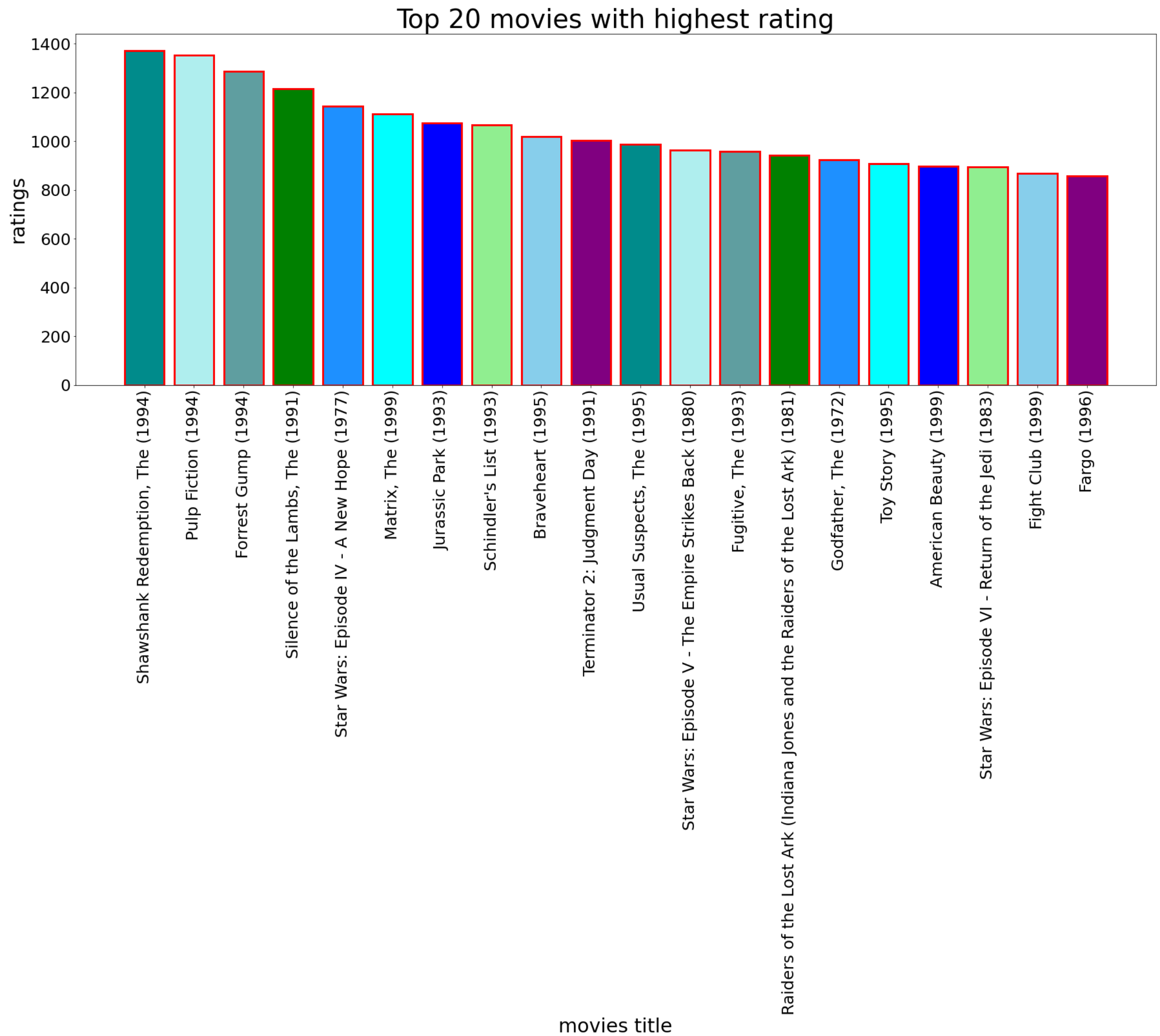
	userId	movieId	rating	timestamp	title	genres
0	1	16	4.0	1217897793	Casino (1995)	Crime|Drama
1	1	24	1.5	1217895807	Powder (1995)	Drama|Sci-Fi
2	1	32	4.0	1217896246	Twelve Monkeys (a.k.a. 12 Monkeys) (1995)	Mystery|Sci-Fi|Thriller
3	1	47	4.0	1217896556	Seven (a.k.a. Se7en) (1995)	Mystery|Thriller
4	1	50	4.0	1217896523	Usual Suspects, The (1995)	Crime|Mystery|Thriller

```python
df_rating=df.groupby(['title'])[['rating']].sum()
high_rated=df_rating.nlargest(20,'rating')
high_rated.head()
```

	rating
title	
Shawshank Redemption, The (1994)	1372.0
Pulp Fiction (1994)	1352.0
Forrest Gump (1994)	1287.0
Silence of the Lambs, The (1991)	1216.5
Star Wars: Episode IV - A New Hope (1977)	1143.5

We then plot a bar graph for identifying the the top 20 movies with highest rating

```python
plt.figure(figsize=(30,10))
plt.title('Top 20 movies with highest rating',fontsize=40)
colors=['darkcyan','paleturquoise','cadetblue','green','dodgerblue','cyan','blue','lightgreen','skyblue','purple']
plt.ylabel('ratings',fontsize=30)
plt.xticks(fontsize=25,rotation=90)
plt.xlabel('movies title',fontsize=30)
plt.yticks(fontsize=25)
plt.bar(high_rated.index,high_rated['rating'],linewidth=3,edgecolor='red',color=colors)
```



Top 20 movies with highest rating
ratings
1400
1200
1000
800
600
400
200
0
movies title
Shawshank Redemption, The (1994)
Pulp Fiction (1994)
Forrest Gump (1994)
Silence of the Lambs, The (1991)
Star Wars: Episode IV - A New Hope (1977)
Matrix, The (1999)
Jurassic Park (1993)
Schindler's List (1993)
Braveheart (1995)
Terminator 2: Judgment Day (1991)
Usual Suspects, The (1995)
Star Wars: Episode V - The Empire Strikes Back (1980)
Fugitive, The (1993)
Raiders of the Lost Ark (Indiana Jones and the Raiders of the Lost Ark) (1981)
Godfather, The (1972)
Toy Story (1995)
American Beauty (1999)
Star Wars: Episode VI - Return of the Jedi (1983)
Fight Club (1999)
Fargo (1996)

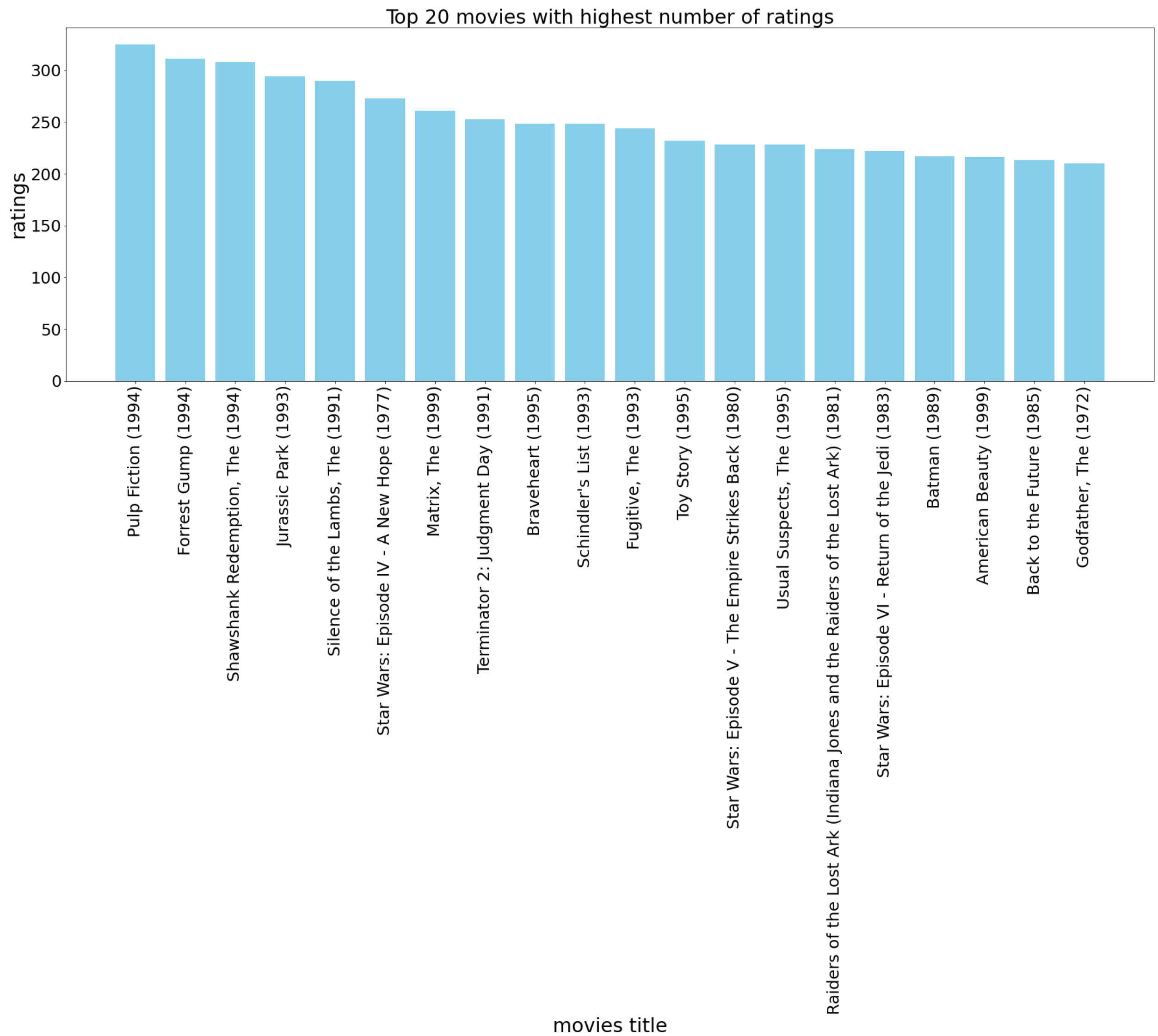
```python
df_rating1=df.groupby('title')[['rating']].count()
rating_count_20=df_rating1.nlargest(20,'rating')
rating_count_20.head()
```

title	rating
Pulp Fiction (1994)	325
Forrest Gump (1994)	311
Shawshank Redemption, The (1994)	308
Jurassic Park (1993)	294
Silence of the Lambs, The (1991)	290

```python
plt.figure(figsize=(30,10))
plt.title('Top 20 movies with highest number of ratings',fontsize=30)
plt.xticks(fontsize=25,rotation=90)
plt.yticks(fontsize=25)
plt.xlabel('movies title',fontsize=30)
plt.ylabel('ratings',fontsize=30)

plt.bar(rating_count_20.index,rating_count_20.rating,color='skyblue')
```

We then plot a bar graph for identifying the top 20 movies with the highest number of ratings.



Pre-processing:

Preprocessing movies dataframe

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.

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

Using regular expressions to find a year stored between parentheses
We specify the parentheses so we don't conflict with movies that have years in their titles

```
movies['year'] = movies.title.str.extract('(\d\d\d\d)',expand=False)
movies['year']
```

```
0        1995
1        1995
2        1995
3        1995
4        1995
         ...
10324    2015
10325    1966
10326    2015
10327    2015
10328    2015
Name: year, Length: 10329, dtype: object
```

```
#Removing the years from the 'title' column

movies['title'] = movies.title.str.replace('(\(\d\d\d\d\))', '')
movies['title']
```

```
0                          Toy Story
1                            Jumanji
2                   Grumpier Old Men
3                  Waiting to Exhale
4        Father of the Bride Part II
                     ...
10324          Cosmic Scrat-tastrophe
10325             Le Grand Restaurant
10326         A Very Murray Christmas
10327                   The Big Short
10328    Marco Polo: One Hundred Eyes
Name: title, Length: 10329, dtype: object
```

We are then applying the strip function to get rid of any ending whitespace characters that may have appeared.

```
#Applying the strip function to get rid of any ending whitespace characters that may have appeared

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

Dropping the attributes which do not provide us with any information

```python
movies= movies.drop('genres', 1)
movies.head()
```

	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

Preprocessing ratings dataframe

```python
ratings = ratings.drop('timestamp', 1)
ratings.head()
```

	userId	movieId	rating
0	1	16	4.0
1	1	24	1.5
2	1	32	4.0
3	1	47	4.0
4	1	50	4.0

Neighborhood Based Collaborative Filtering

Collaborative filtering is the most common technique when it comes to recommender systems. As its name suggests, it is a technique that helps filter out items for a user in a collaborative way, that is, based on the preferences of similar users.

Memory-based or neighborhood-based methods use user rating historical data to compute the similarity between users or items. The idea behind these methods is to define a similarity measure between users or items, and find the most similar to recommend unseen items.

We begin by creating an input user to recommend movies to

```python
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

```python
#Filtering out the movies by title
inputId = movies[movies['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)

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

With the movie ID's in our input, we can now get the subset of users that have watched and reviewed the movies in our input.

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

userSubset = ratings[ratings['movieId'].isin(inputMovies['movieId'].tolist())]
userSubset.head()
```

	userId	movieId	rating
15	1	296	4.0
113	2	1	5.0
166	3	296	5.0
220	4	296	4.0
339	5	1	4.0

```python
# We now group up the rows by user ID
userSubsetGroup = userSubset.groupby(['userId'])
```

```python
userSubsetGroup.get_group(220)
```

	userId	movieId	rating
30252	220	1	4.0
30253	220	2	3.5
30326	220	296	4.0
30635	220	1968	3.5

```python
len(userSubsetGroup.get_group(220))
```

4

```python
userSubsetGroup.get_group(100)
```

	userId	movieId	rating
11287	100	1	3.0

```python
len(userSubsetGroup.get_group(100))
```

1

We 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)
```

```
#Top most user with id 62 having all 5 similar moves watched
userSubsetGroup[0]
```

```
(62,
       userId  movieId  rating
 5535      62        1     2.0
 5536      62        2     1.5
 5604      62      296     5.0
 5760      62     1274     3.5
 5857      62     1968     1.0)
```

```
#id of top user group
userSubsetGroup[0][0]
```

```
62
```

```
#dataframe of top user group
userSubsetGroup[0][1]
```

	userId	movieId	rating
5535	62	1	2.0
5536	62	2	1.5
5604	62	296	5.0
5760	62	1274	3.5
5857	62	1968	1.0

Similarity of users to input user

We are now going to compare all users 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.

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 .

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

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.

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

```python
#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
    group = group.sort_values(by='movieId')
    inputMovies = inputMovies.sort_values(by='movieId')

    #Get the N (total similar movies watched) 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())]
    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 Sxx != 0 and Syy != 0:
        pearsonCorrelationDict[name] = Sxy/np.sqrt(Sxx*Syy)
    else:
        pearsonCorrelationDict[name] = 0
```

```python
pearsonCorrelationDict.items()
```

Python

```
dict_items([(62, 0.44965838938680786), (122, 0.8770580193070289), (224, 0.29417420270727607), (409, 0.8056292332943623), (451, 0.6564386345361464), (461, 0.11720
```

```python
pearsonDF = pd.DataFrame.from_dict(pearsonCorrelationDict, orient='index')
pearsonDF.head()
```

Python

	0
62	0.449658
122	0.877058
224	0.294174
409	0.805629
451	0.656439

```python
pearsonDF.columns = ['similarityIndex']
pearsonDF['userId'] = pearsonDF.index
pearsonDF.index = range(len(pearsonDF))
pearsonDF.head()
```

Python

	similarityIndex	userId
0	0.449658	62
1	0.877058	122
2	0.294174	224
3	0.805629	409
4	0.656439	451

The top x similar users to input user

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

	similarityIndex	userId
50	1.000000	158
53	1.000000	213
94	0.987829	615
55	0.987829	228
39	0.944911	44

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 these two tables.

```python
topUsersRating = topUsers.merge(ratings, left_on='userId', right_on='userId', how='inner')
topUsersRating.head()
```

	similarityIndex	userId	movieId	rating
0	1.0	158	1	4.5
1	1.0	158	2	4.0
2	1.0	158	6	4.5
3	1.0	158	10	4.0
4	1.0	158	32	4.0

We now need to 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.

```python
topUsersRating['weightedRating'] = topUsersRating['similarityIndex']*topUsersRating['rating']
topUsersRating.head()
```

	similarityIndex	userId	movieId	rating	weightedRating
0	1.0	158	1	4.5	4.5
1	1.0	158	2	4.0	4.0
2	1.0	158	6	4.5	4.5
3	1.0	158	10	4.0	4.0
4	1.0	158	32	4.0	4.0

```python
#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	30.930292	115.143135
2	27.603656	85.617531
3	5.268273	16.472588
4	1.255929	3.767787
5	5.531023	17.606041

```python
recommendation_df = pd.DataFrame()
#We take the weighted average
recommendation_df['Weighted average recommendation score'] = tempTopUsersRating['sum_weightedRating']/tempTopUsersRating['sum_simi
recommendation_df['movieId'] = tempTopUsersRating.index
recommendation_df.head()
```

Python

	Weighted average recommendation score	movieId
movieId		
1	3.722666	1
2	3.101674	2
3	3.126753	3
4	3.000000	4
5	3.183144	5

+ Code + Markdown

Recommended movies

```python
recommendation_df = recommendation_df.sort_values(by='Weighted average recommendation score', ascending=False)
recommendation_df.head()
```

	Weighted average recommendation score	movieId
movieId		
27366	5.0	27366
1099	5.0	1099
897	5.0	897
101529	5.0	101529
68522	5.0	68522

```python
movies.loc[movies['movieId'].isin(recommendation_df.head(20)['movieId'].tolist())]
```

	movieId	title	year
718	897	For Whom the Bell Tolls	1943
895	1099	Christmas Carol, A	1938
996	1236	Trust	1990
1486	1914	Smoke Signals	1998
1768	2227	Lodger: A Story of the London Fog, The	1927
3485	4454	More	1998
4888	6672	War Photographer	2001
5152	7075	Court Jester, The	1956
5244	7215	To Have and Have Not	1944
5258	7234	Strada, La	1954
6307	27366	Werckmeister Harmonies (Werckmeister harmóniák)	2000
7634	58078	Air I Breathe, The	2007
7742	59810	Recount	2008
7868	62049	1984	1984
7897	62644	Wave, The (Welle, Die)	2008
7973	65188	Dear Zachary: A Letter to a Son About His Father	2008
8102	68522	Earth	2007
9634	101529	Brass Teapot, The	2012
9641	101862	50 Children: The Rescue Mission of Mr. And Mrs...	2013
9754	104177	From One Second to the Next	2013

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

Content Based Recommendation

Content-based filtering is a type of recommender system that attempts to guess what a user may like based on that user’s activity.

Content-based filtering makes recommendations by using keywords and attributes assigned to objects in a database (e.g., items in an online marketplace) and matching them to a user profile. The user profile is created based on data derived from a user’s actions, such as purchases, ratings (likes and dislikes), downloads, items searched for on a website and/or placed in a cart, and clicks on product links.

TF-IDF

Term frequency-inverse document frequency is a text vectorizer that transforms the text into a usable vector. It combines 2 concepts, Term Frequency (TF) and Document Frequency (DF).

The term frequency is the number of occurrences of a specific term in a document. Term frequency indicates how important a specific term is in a document. Term frequency represents every text from the data as a matrix whose rows are the number of documents and columns are the number of distinct terms throughout all documents.

Document frequency is the number of documents containing a specific term. Document frequency indicates how common the term is.

Inverse document frequency (IDF) is the weight of a term, it aims to reduce the weight of a term if the term’s occurrences are scattered throughout all the documents. IDF can be calculated as follow:

```python
cv=TfidfVectorizer()
tfidf_matrix=cv.fit_transform(movies_df['genres'])
```

Python

```python
movie_user = df.pivot_table(index='userId',columns='title',values='rating')
movie_user.head()
```

Python

title	'71 (2014)	'Hellboy': The Seeds of Creation (2004)	'Round Midnight (1986)	'Til There Was You (1997)	'burbs, The (1989)	'night Mother (1986)	(500) Days of Summer (2009)	*batteries not included (1987)	...And Justice for All (1979)	10 (1979)	...	[REC] (2007)	[REC]² (2009)	[REC]³ 3 Génesis (2012)	a/k/a Tommy Chong (2005)	eXistenZ (1999)	loudQUIETloud: A Film About the Pixies (2006)	(20
userId																		
1	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	N
2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	N
3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	N
4	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	N
5	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	N

5 rows × 10323 columns

```python
cosine_sim = linear_kernel(tfidf_matrix, tfidf_matrix)
```

```python
indices=pd.Series(movies_df.index,index=movies_df['title'])
titles=movies_df['title']
def recommendations(title):
    idx = indices[title]
    sim_scores = list(enumerate(cosine_sim[idx]))
    sim_scores = sorted(sim_scores, key=lambda x: x[1], reverse=True)
    sim_scores = sim_scores[1:21]
    movie_indices = [i[0] for i in sim_scores]
    return titles.iloc[movie_indices]
```

Recommended movies

```
recommendations('Toy Story 2 (1999)')
```

```
1815                                                Antz (1998)
2496                                         Toy Story 2 (1999)
2967         Adventures of Rocky and Bullwinkle, The (2000)
3166                          Emperor's New Groove, The (2000)
3811                                    Monsters, Inc. (2001)
6617      DuckTales: The Movie - Treasure of the Lost La...
6997                                          Wild, The (2006)
7382                                    Shrek the Third (2007)
7987                            Tale of Despereaux, The (2008)
9215      Asterix and the Vikings (Astérix et les Viking...
9732                                              Turbo (2013)
10052                                     Boxtrolls, The (2014)
1595                                 Black Cauldron, The (1985)
1675                                Lord of the Rings, The (1978)
2696                      We're Back! A Dinosaur's Story (1993)
3420                          Atlantis: The Lost Empire (2001)
3535                               Land Before Time, The (1988)
4314      Pokemon 4 Ever (a.k.a. Pokémon 4: The Movie) (...
4799                    Sinbad: Legend of the Seven Seas (2003)
5539                               Phantom Tollbooth, The (1970)
Name: title, dtype: object
```

Advantages and Disadvantages of Content-Based Filtering

Advantages

- Learns user's preferences
- Highly personalized for the user

Disadvantages

- Doesn't take into account what others think of the item, so low quality item recommendations might happen
- Extracting data is not always intuitive
- Determining what characteristics of the item the user dislikes or likes is not always obvious

Evaluation Metrics

```
df = pd.read_csv('ratings.csv',
                 error_bad_lines=False,
                 warn_bad_lines=False,
                 skiprows=lambda i: i>0 and random.random() > 0.2)

print(len(df))
print(df['rating'].unique().tolist())
print(len(df['userId'].unique().tolist()))
print(len(df['movieId'].unique().tolist()))
```

```
21141
[4.0, 0.5, 3.0, 3.5, 5.0, 1.5, 4.5, 2.5, 2.0, 1.0]
666
5456
```

```
reader = Reader(rating_scale=(0,10)) # rating scale range
data = Dataset.load_from_df(df[['userId', 'movieId', 'rating']], reader)
print(type(data))
```

```
<class 'surprise.dataset.DatasetAutoFolds'>
```

```
trainset, testset = train_test_split(data, test_size=0.25)
print(type(trainset))
```

```
<class 'surprise.trainset.Trainset'>
```

```
algo = SVD()
algo.fit(trainset)
```

Mean Squared Error (MSE)

MSE is one of the most common regression loss functions. In Mean Squared Error also known as L2 loss, we calculate the error by squaring the difference between the predicted value and actual value and averaging it across the dataset. MSE is also known as Quadratic loss as the penalty is not proportional to the error but to the square of the error. Squaring the error gives higher weight to the outliers, which results in a smooth gradient for small errors. Optimization algorithms benefit from this penalization for large errors as it is helpful in finding the optimum values for parameters.

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Root Mean Squared Error (RMSE)

RMSE is computed by taking the square root of MSE. RMSE is also called the Root Mean Square Deviation. It measures the average magnitude of the errors and is concerned with the deviations from the actual value. RMSE value with zero indicates that the model has a perfect fit. The lower the RMSE, the better the model and its predictions. A higher RMSE indicates that there is a large deviation from the residual to the ground truth. RMSE can be used with different features as it helps in figuring out if the feature is improving the model's prediction or not.

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

```
predictions = algo.test(testset)
```

```
accuracy.mse(predictions)
```

```
MSE: 0.8516

0.8515791165180502
```

```
accuracy.rmse(predictions)
```

```
RMSE: 0.9228

0.9228104445215443
```

+ Code + Markdown