MOVIE RECOMMENDATION SYSTEM-PYTHON

ABSTRACT

In this package we'll be building a baseline Movie Recommendation System using TMDB 5000 Movie Dataset.There are basically three types of recommender systems:-

- **Demographic Filtering**- They offer generalized recommendations to every user, based on movie popularity and/or genre. The System recommends the same movies to users with similar demographic features. The basic idea behind this system is that movies that are more popular and critically acclaimed will have a higher probability of being liked by the average audience.

- **Content Based Filtering**- They suggest similar items based on a particular item. This system uses item metadata, such as genre, director, description, actors, etc. for movies, to make these recommendations. The general idea behind these recommender systems is that if a person liked a particular item, he or she will also like an item that is similar to it.

- **Collaborative Filtering**- This system matches persons with similar interests and provides recommendations based on this matching. Collaborative filters do not require item metadata like its content-based counterparts.

The first dataset contains the following features:-

- movie_id - A unique identifier for each movie.
- cast - The name of lead and supporting actors.
- crew - The name of Director, Editor, Composer, Writer etc.

The second dataset has the following features:-

- budget - The budget in which the movie was made.
- genre - The genre of the movie, Action, Comedy ,Thriller etc.
- homepage - A link to the homepage of the movie.

- id - This is infact the movie_id as in the first dataset.
- keywords - The keywords or tags related to the movie.
- original_language - The language in which the movie was made.
- original_title - The title of the movie before translation or adaptation.
- overview - A brief description of the movie.
- popularity - A numeric quantity specifying the movie popularity.
- production_companies - The production house of the movie.
- production_countries - The country in which it was produced.

- release_date - The date on which it was released.
- revenue - The worldwide revenue generated by the movie.
- runtime - The running time of the movie in minutes.
- status - "Released" or "Rumored".
- tagline - Movie's tagline.
- title - Title of the movie.
- vote_average - average ratings the movie received.
- vote_count - the count of votes received.

INTRODUCTION

Let's upload two datasets and join the datasets based on the 'id' column.

```
df1=pd.read_csv('/content/tmdb_5000_credits.csv.zip')
df2=pd.read_csv('/content/tmdb_5000_movies.csv')
```

```
df1.columns = ['id','tittle','cast','crew']
df2= df2.merge(df1,on='id')
```

1. DEMOGRAPHIC FILTERING:

First,

- We need a metric to score or rate movie
- Calculate the score for every movie
- Sort the scores and recommend the best rated movie to the users.

So the method of Weighted Rating is used:

$$\text{Weighted Rating (WR)} = \left(\frac{v}{v+m}.R\right) + \left(\frac{m}{v+m}.C\right)$$

where,

- v is the number of votes for the movie;
- m is the minimum votes required to be listed in the chart;
- R is the average rating of the movie; And
- C is the mean vote across the whole report

C is calculated ,

```
C= df2['vote_average'].mean()
print(C)
```

C is 6.092171559442011

So, the mean rating for all the movies is approx 6 on a scale of 10.

The next step is to determine an appropriate value for m,

```
m= df2['vote_count'].quantile(0.9)
print(m)
```

```
1838.4000000000015
```

We will use 90th percentile as our cutoff. It must have more votes than at least 90% of the movies in the list.

Now, we can filter out the movies that qualify for the chart

```
q_movies = df2.copy().loc[df2['vote_count'] >= m]
q_movies.shape
```

We see that there are 481 movies which qualify to be in this list.

To calculate our metric for each qualified movie a function weighed_rating() is used.

```
def weighted_rating(x, m=m, C=C):
    v = x['vote_count']
    R = x['vote_average']
    # Calculation based on the IMDB formula
    return (v/(v+m) * R) + (m/(m+v) * C)
```

A new feature **score**, of which we'll calculate the value by applying this function to our DataFrame of qualified movies.

```
def weighted_rating(x, m=m, C=C):
    v = x['vote_count']
    R = x['vote_average']
    # Calculation based on the IMDB formula
    return (v/(v+m) * R) + (m/(m+v) * C)
```

```
# Define a new feature 'score' and calculate its value with `weighted_rating()`
q_movies['score'] = q_movies.apply(weighted_rating, axis=1)
```

Finally, let's sort the Dataframe based on the score feature and output the title, vote count, vote average and weighted rating or score of the top 15 movies.

```
#Sort movies based on score calculated above
q_movies = q_movies.sort_values('score', ascending=False)

#Print the top 15 movies
q_movies[['title', 'vote_count', 'vote_average', 'score']].head(15)
```

	title	vote_count	vote_average	score
1881	The Shawshank Redemption	8205	8.5	8.059258
662	Fight Club	9413	8.3	7.939256
65	The Dark Knight	12002	8.2	7.920020
3232	Pulp Fiction	8428	8.3	7.904645
96	Inception	13752	8.1	7.863239
3337	The Godfather	5893	8.4	7.851236
95	Interstellar	10867	8.1	7.809479
809	Forrest Gump	7927	8.2	7.803188
329	The Lord of the Rings: The Return of the King	8064	8.1	7.727243
1990	The Empire Strikes Back	5879	8.2	7.697884

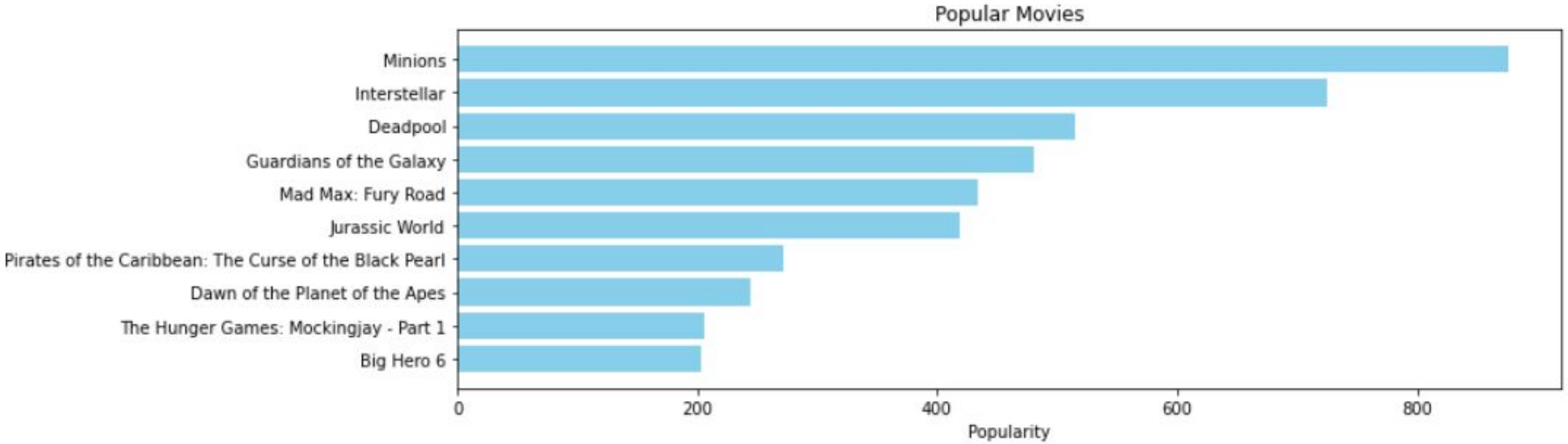
The **Trending Now** tab of these systems we find movies that are very popular and they can just be obtained by sorting the dataset by the popularity column.

```
pop= df2.sort_values('popularity', ascending=False)
import matplotlib.pyplot as plt
plt.figure(figsize=(12,4))

plt.barh(pop['title'].head(10),pop['popularity'].head(10), align='center',color='skyblue')
plt.gca().invert_yaxis()
plt.xlabel("Popularity")
plt.title("Popular Movies")
```

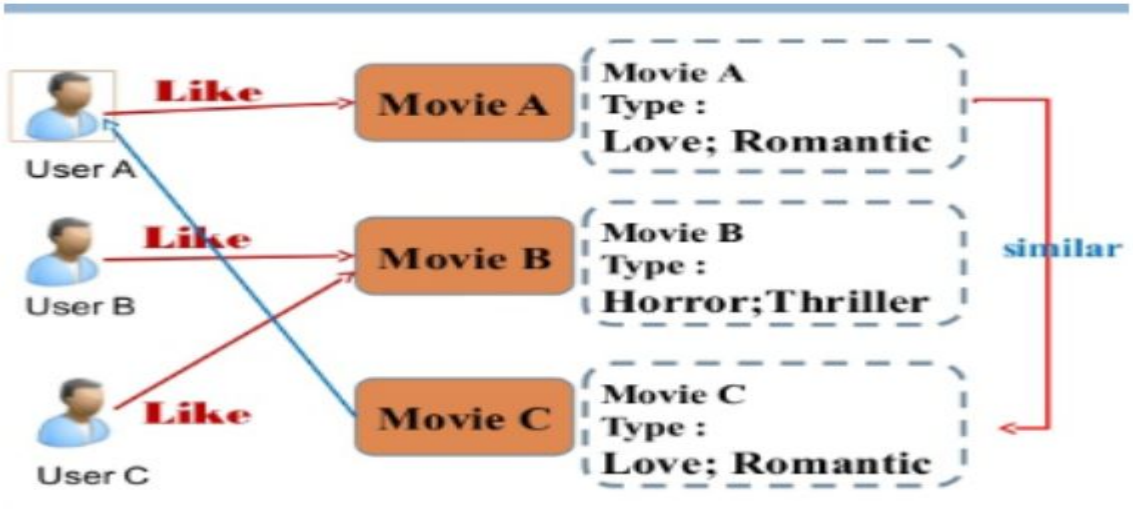
Demographic recommender provides a general chart of recommended movies to all the users.It is not based on the taste and interest of a single user.

```
Text(0.5, 1.0, 'Popular Movies')
```



2.CONTENT BASED FILTERING:

In this system the content of the movie (overview, cast, crew, keyword, tagline etc) is used to find its similarity with other movies. Then the movies that are most likely to be similar are recommended.



- # PLOT DESCRIPTION BASED RECOMMENDER:

A similarity score is computed for all movies and the movies that have similar scores are recommended.Plot description is available in the overview feature of the dataset.

```
#Plot description based Recommender
df2['overview'].head(5)
```

```
0    In the 22nd century, a paraplegic Marine is di...
1    Captain Barbossa, long believed to be dead, ha...
2    A cryptic message from Bond’s past sends him o...
3    Following the death of District Attorney Harve...
4    John Carter is a war-weary, former military ca...
Name: overview, dtype: object
```

Now we'll compute Term Frequency-Inverse Document Frequency (TF-IDF) vectors for each overview.

->TERM FREQUENCY:

It is the relative frequency of a word in a document and is given as **(term instances/total instances)**.

->INVERSE DOCUMENT FREQUENCY:

It is the relative count of documents containing the term is given as **log(number of documents/documents with term)** .The overall importance of each word to the documents in which they appear is equal to **TF * IDF**

This will give us a matrix where each column represents a word in the overview vocabulary (all the words that appear in at least one document) and each column represents a movie, as before.

Scikit-learn gives us a built-in TfIdfVectorizer class that produces the TF-IDF matrix in a couple of lines.

```
#Import TfIdfVectorizer from scikit-learn
from sklearn.feature_extraction.text import TfidfVectorizer

#Define a TF-IDF Vectorizer Object. Remove all english stop words such as 'the', 'a'
tfidf = TfidfVectorizer(stop_words='english')

#Replace NaN with an empty string
df2['overview'] = df2['overview'].fillna('')

#Construct the required TF-IDF matrix by fitting and transforming the data
tfidf_matrix = tfidf.fit_transform(df2['overview'])

#Output the shape of tfidf_matrix
tfidf_matrix.shape
```

```
(4803, 20978)
```

We see that over 20,000 different words were used to describe the 4800 movies in our dataset.

With this matrix in hand, we can now compute a similarity score. We use the cosine similarity score to calculate a numeric quantity that denotes the similarity between two movies.Since we have used the TF-IDF vectorizer, calculating the dot product will directly give us the cosine similarity score.

Therefore, we will use sklearn's linear_kernel() .

```
# Import linear_kernel
from sklearn.metrics.pairwise import linear_kernel

# Compute the cosine similarity matrix
cosine_sim = linear_kernel(tfidf_matrix, tfidf_matrix)
```

We are going to define a function that takes in a movie title as an input and outputs a list of the 10 most similar movies.

```
# Function that takes in movie title as input and outputs most similar movies
def get_recommendations(title, cosine_sim=cosine_sim):
    # Get the index of the movie that matches the title
    idx = indices[title]

    # Get the pairwsie similarity scores of all movies with that movie
    sim_scores = list(enumerate(cosine_sim[idx]))

    # Sort the movies based on the similarity scores
    sim_scores = sorted(sim_scores, key=lambda x: x[1], reverse=True)

    # Get the scores of the 10 most similar movies
    sim_scores = sim_scores[1:11]

    # Get the movie indices
    movie_indices = [i[0] for i in sim_scores]

    # Return the top 10 most similar movies
    return df2['title'].iloc[movie_indices]
```

These are the following steps we followed :-

- Get the index of the movie given its title.
- Get the list of cosine similarity scores for that particular movie with all movies. Convert it into a list of tuples where the first element is its position and the second is the similarity score.
- Sort the aforementioned list of tuples based on the similarity scores; that is, the second element.
- Get the top 10 elements of this list. Ignore the first element as it refers to self (the movie most similar to a particular movie is the movie itself).
- Return the titles corresponding to the indices of the top elements.

```
mov=input("Enter the movie name")
```

```
Enter the movie nameThe Avengers
```

```
get_recommendations(mov)
```

```
7                  Avengers: Age of Ultron
3144                                Plastic
1715                                Timecop
4124                    This Thing of Ours
3311                 Thank You for Smoking
3033                          The Corruptor
588      Wall Street: Money Never Sleeps
2136           Team America: World Police
1468                          The Fountain
1286                           Snowpiercer
Name: title, dtype: object
```

- # CREDITS,GENRES AND KEYWORDS BASED RECOMMENDER:

We are going to build a recommender based on the following metadata: the 3 top actors, the director, related genres and the movie plot keywords.From the cast, crew and keywords features, we need to extract the three most important actors, the director and the keywords associated with that movie.

Functions that will help us to extract the required information from each feature.

```
# Get the director's name from the crew feature. If director is not listed, return NaN
def get_director(x):
    for i in x:
        if i['job'] == 'Director':
            return i['name']
    return np.nan
```

```python
# Returns the list top 3 elements or entire list; whichever is more.
def get_list(x):
  if isinstance(x, list):
      names = [i['name'] for i in x]
      #Check if more than 3 elements exist. If yes, return only first three. If no, return entire list.
      if len(names) > 3:
          names = names[:3]
      return names

  #Return empty list in case of missing/malformed data
  return []
```

```python
# Define new director, cast, genres and keywords features that are in a suitable form.
df2['director'] = df2['crew'].apply(get_director)

features = ['cast', 'keywords', 'genres']
for feature in features:
    df2[feature] = df2[feature].apply(get_list)
```

```python
# Print the new features of the first 3 films
df2[['title', 'cast', 'director', 'keywords', 'genres']].head()
```

	title	cast	director	keywords	genres
0	Avatar	[Sam Worthington, Zoe Saldana, Sigourney Weaver]	James Cameron	[culture clash, future, space war]	[Action, Adventure, Fantasy]
1	Pirates of the Caribbean: At World's End	[Johnny Depp, Orlando Bloom, Keira Knightley]	Gore Verbinski	[ocean, drug abuse, exotic island]	[Adventure, Fantasy, Action]
2	Spectre	[Daniel Craig, Christoph Waltz, Léa Seydoux]	Sam Mendes	[spy, based on novel, secret agent]	[Action, Adventure, Crime]
3	The Dark Knight Rises	[Christian Bale, Michael Caine, Gary Oldman]	Christopher Nolan	[dc comics, crime fighter, terrorist]	[Action, Crime, Drama]
4	John Carter	[Taylor Kitsch, Lynn Collins, Samantha Morton]	Andrew Stanton	[based on novel, mars, medallion]	[Action, Adventure, Science Fiction]

The next step would be to convert the names and keyword instances into lowercase and strip all the spaces between them. This is done so that our vectorizer doesn't count the Johnny of "Johnny Depp" and "Johnny Galecki" as the same.

```
# Function to convert all strings to lower case and strip names of spaces
def clean_data(x):
    if isinstance(x, list):
        return [str.lower(i.replace(" ", "")) for i in x]
    else:
        #Check if director exists. If not, return empty string
        if isinstance(x, str):
            return str.lower(x.replace(" ", ""))
        else:
            return ''
```

```
# Apply clean_data function to your features.
features = ['cast', 'keywords', 'director', 'genres']

for feature in features:
    df2[feature] = df2[feature].apply(clean_data)
```

Create our "metadata soup", which is a string that contains all the metadata that we want to feed to our vectorizer (namely actors, director and keywords).

```
def create_soup(x):
    return ' '.join(x['keywords']) + ' ' + ' '.join(x['cast']) + ' ' + x['director'] + ' ' + ' '.join(x['genres'])
df2['soup'] = df2.apply(create_soup, axis=1)
```

The next steps are the same as what we did with our plot description based recommender. One important difference is that we use the **CountVectorizer()** instead of TF-IDF.

```
# Import CountVectorizer and create the count matrix
from sklearn.feature_extraction.text import CountVectorizer

count = CountVectorizer(stop_words='english')
count_matrix = count.fit_transform(df2['soup'])
```

```
# Compute the Cosine Similarity matrix based on the count_matrix
from sklearn.metrics.pairwise import cosine_similarity

cosine_sim2 = cosine_similarity(count_matrix, count_matrix)
```

```
# Reset index of our main DataFrame and construct reverse mapping as before
df2 = df2.reset_index()
indices = pd.Series(df2.index, index=df2['title'])
```

```
mov=input("Enter the movie name")
```

```
Enter the movie nameThe Avengers
```

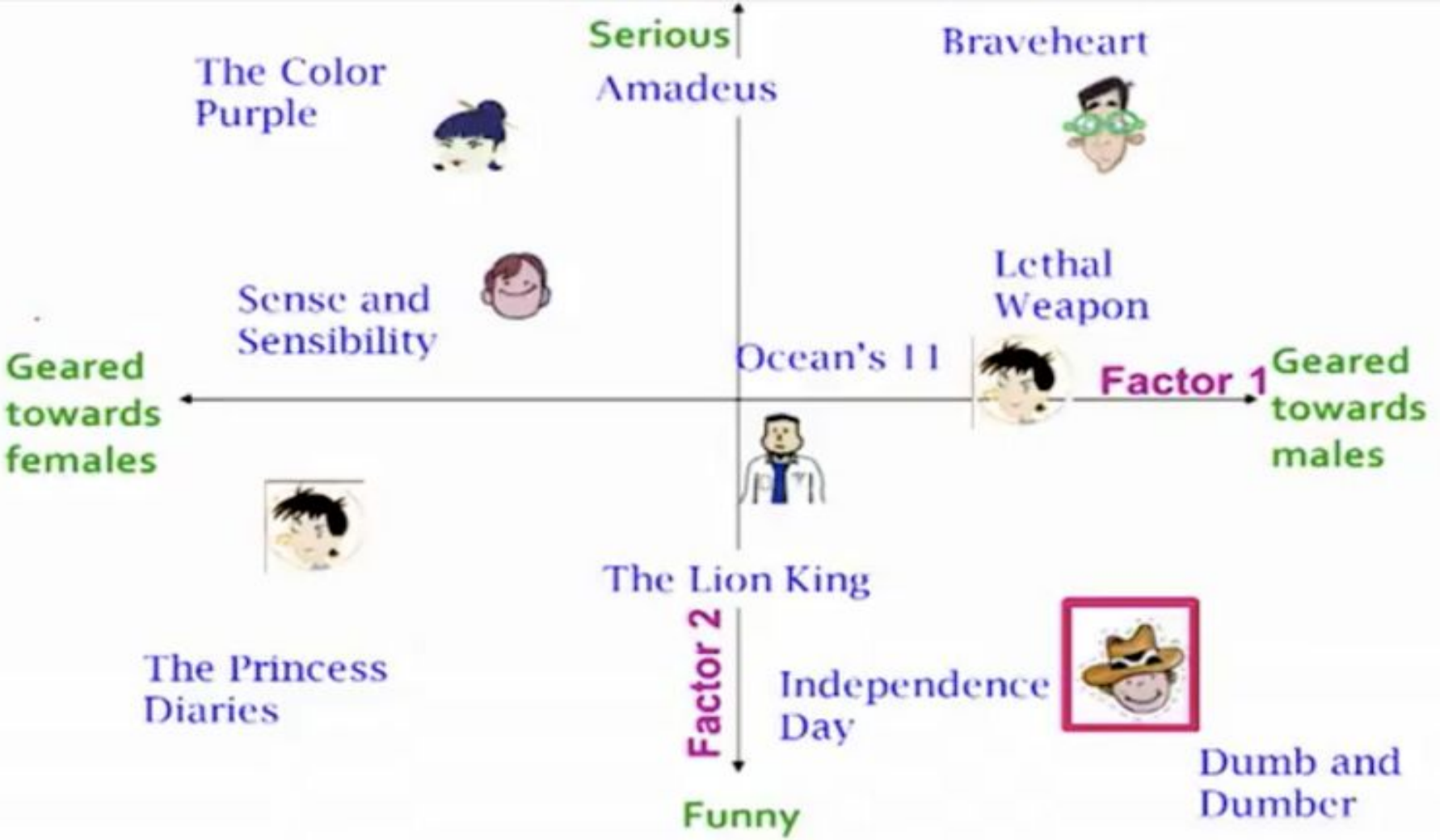
```
get_recommendations(mov)
```

```
7                Avengers: Age of Ultron
3144                              Plastic
1715                              Timecop
4124                 This Thing of Ours
3311              Thank You for Smoking
3033                       The Corruptor
588     Wall Street: Money Never Sleeps
2136          Team America: World Police
1468                         The Fountain
1286                          Snowpiercer
Name: title, dtype: object
```

3.COLLABORATIVE FILTERING

- SINGLE VALUE DECOMPOSITON

  This method uses latent factor model to capture the similarity between users and items. One common metric is Root Mean Square Error (RMSE). The lower the RMSE, the better the performance .Latent model is a broad idea which describes a property or concept that a user or an item have.Essentially, we map each user and each item into a latent space with dimension r. Therefore, it helps us better understand the relationship between users and items as they become directly comparable.


Serious
Amadeus
The Color Purple
Braveheart
Sense and Sensibility
Lethal Weapon
Ocean's 11
Geared towards females
Factor 1
Geared towards males
The Lion King
Factor 2
The Princess Diaries
Independence Day
Dumb and Dumber
Funny

Since the dataset we used before did not have userId(which is necessary for collaborative filtering) let's load another dataset. We'll be using the Surprise library to implement SVD.

```
#COLLABORATIVE FILTERING
#Single value decomposition
!pip install surprise
from surprise import Reader, Dataset, SVD
reader = Reader()
ratings = pd.read_csv('/content/ratings_small.csv.zip')
ratings.head()
```

	userId	movieId	rating	timestamp
0	1	31	2.5	1260759144
1	1	1029	3.0	1260759179
2	1	1061	3.0	1260759182
3	1	1129	2.0	1260759185
4	1	1172	4.0	1260759205

Note that in this dataset movies are rated on a scale of 5 unlike the earlier one.

```
data = Dataset.load_from_df(ratings[['userId', 'movieId', 'rating']], reader)
svd = SVD()
```

```
trainset = data.build_full_trainset()
svd.fit(trainset)
```

We get a mean Root Mean Sqaure Error of 0.89 approx which is more than good enough for our case. Let us now train on our dataset and arrive at predictions.

Let us pick user with user Id 1 and check the ratings she/he has given.

```
ratings[ratings['userId'] == 1]
```

	userId	movieId	rating	timestamp
0	1	31	2.5	1260759144
1	1	1029	3.0	1260759179
2	1	1061	3.0	1260759182
3	1	1129	2.0	1260759185
4	1	1172	4.0	1260759205
5	1	1263	2.0	1260759151
6	1	1287	2.0	1260759187

```
svd.predict(1, 302, 3)
```

```
Prediction(uid=1, iid=302, r_ui=3, est=2.6789073048891954, details={'was_impossible': False})
```

For movie with ID 302, we get an estimated prediction of 2.618. One startling feature of this recommender system is that it doesn't care what the movie is (or what it contains). It works purely on the basis of an assigned movie ID and tries to predict ratings based on how the other users have predicted the movie.

CONCLUSION

We created movie recommenders using demographic , content- based and collaborative filtering. While demographic filtering is very elementary and Hybrid Systems can take advantage of content-based and collaborative filtering as the two approaches are proved to be almost complimentary.

THANK YOU!!!

DONE BY-

19PD09-DHIKSHITHA A

19PD38-SWATHI PRATHAA P