Movie Recommender Systems

Recommendation Systems are a type of **information filtering systems** as they improve the quality of search results and provides items that are more relevant to the search item or are related to the search history of the user.

They are used to predict the rating or preference that a user would give to an item. Almost every major tech company has applied them in some form or another: Amazon uses them to suggest products to customers, YouTube employs them to decide which video to play next on autoplay, and Facebook utilizes them to recommend pages to like and people to follow. Furthermore, companies like Netflix and Spotify heavily rely on the effectiveness of their recommendation engines for their business and success.

There are basically three types of recommender systems:

Demographic Filtering: These systems 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. However, since each user is different, this approach is considered too simplistic. The underlying idea 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: This type suggests similar items based on a particular item. It utilizes item metadata, such as genre, director, description, actors, etc., for movies, to make recommendations. The underlying concept of these recommender systems is that if a person liked a particular item, they are likely to appreciate an item that is similar to it.

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

load the data

In [1]:
```
import pandas as pd
import numpy as np
df1=pd.read_csv('../input/tmdb-movie-metadata/tmdb_5000_credits.csv')
df2=pd.read_csv('../input/tmdb-movie-metadata/tmdb_5000_movies.csv')
```

In [2]:
```
df1.columns = ['id','tittle','cast','crew']
df2= df2.merge(df1,on='id')
```

In [3]:
```
df2.head(5)
```

Out[3]:

	budget	genres	homepage	id	keywords	original_language	original_title	overview	popular
0	237000000	[{"id": 28, "name": "Action"}, {"id": 12, "nam...	http://www.avatarmovie.com/	19995	[{"id": 1463, "name": "culture clash"}, {"id":...	en	Avatar	In the 22nd century, a paraplegic Marine is di...	150.437
1	300000000	[{"id": 12, "name": "Adventure"}, {"id": 14, "...	http://disney.go.com/disneypictures/pirates/	285	[{"id": 270, "name": "ocean"}, {"id": 726, "na...	en	Pirates of the Caribbean: At World's End	Captain Barbossa, long believed to be dead, ha...	139.082
2	245000000	[{"id": 28, "name": "Action"}, {"id": 12, "nam...	http://www.sonypictures.com/movies/spectre/	206647	[{"id": 470, "name": "spy"}, {"id": 818, "name...	en	Spectre	A cryptic message from Bond's past sends him o...	107.376
3	250000000	[{"id": 28, "name": "Action"}, {"id": 80, "nam...	http://www.thedarkknightrises.com/	49026	[{"id": 849, "name": "dc comics"}, {"id": 853,...	en	The Dark Knight Rises	Following the death of District Attorney Harve...	112.312
4	260000000	[{"id": 28, "name": "Action"}, {"id": 12, "nam...	http://movies.disney.com/john-carter	49529	[{"id": 818, "name": "based on novel"}, {"id":...	en	John Carter	John Carter is a war-weary, former military ca...	43.926

Demographic Filtering

Before getting started with this -

- 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.

We could use the average ratings of the movie as the score, but this approach wouldn't be fair enough. For instance, a movie with an average rating of 8.9 but only 3 votes cannot be considered better than a movie with a 7.8 average rating but 40 votes. Therefore, I'll be employing IMDB's weighted rating (wr), which is calculated as follows:

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

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;
- C is the mean vote across the whole report

We already have v(**vote_count**) and R (**vote_average**) and C can be calculated as

In [4]:
```
C= df2['vote_average'].mean()
C
```

Out[4]:
```
6.092171559442011
```

So, the mean rating for all the movies is approximately 6 on a scale of 10. The subsequent step involves determining an appropriate value for 'm,' representing the minimum votes required for a movie to be listed in the chart. We'll utilize the 90th percentile as our cutoff. In essence, for a movie to be featured in the charts, it must garner more votes than at least 90% of the movies in the list.

In [5]:
```
m= df2['vote_count'].quantile(0.9)
m
```

Out[5]:
```
1838.4000000000015
```

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

In [6]:
```
q_movies = df2.copy().loc[df2['vote_count'] >= m]
q_movies.shape
```

Out[6]:
```
(481, 23)
```

Having identified 481 movies that qualify to be in this list, our next step involves calculating our metric for each qualified movie. To achieve this, we'll create a function called 'weighted_rating()' and introduce a new feature called 'score.' We'll calculate the value of this new feature by applying the 'weighted_rating()' function to our DataFrame of qualified movies.

In [7]:
```
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)
```

In [8]:
```
# Define a new feature 'score' and calculate its value with `weighted_rating()`
q_movies['score'] = q_movies.apply(weighted_rating, axis=1)
```

Lastly, we'll sort the DataFrame based on the 'score' feature and display the titles, vote count, vote average, and weighted rating (score) of the top 10 movies.

In [9]:
```
#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(10)
```

Out[9]:

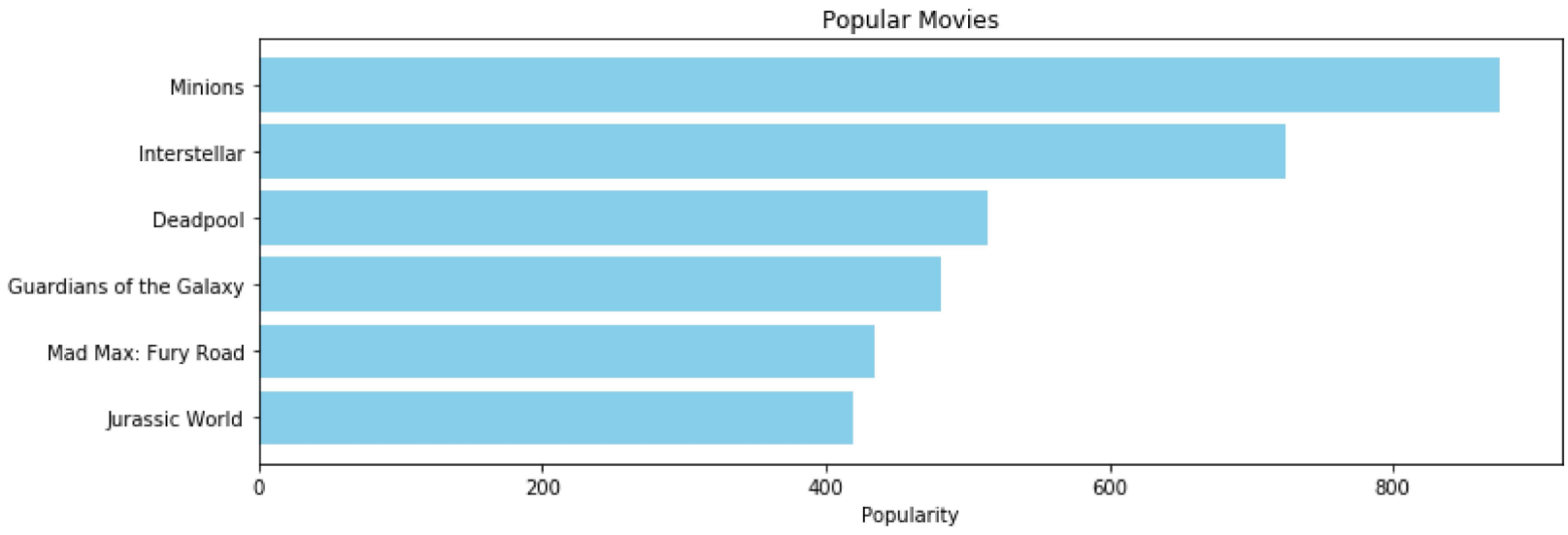
	title	vote_count	vote_average	score
1881	The Shawshank Redemption	8205	8.5	8.059258
662	Fight Club	9413	8.3	7.939256
65	The Dark Knight	12002	8.2	7.920020
3232	Pulp Fiction	8428	8.3	7.904645
96	Inception	13752	8.1	7.863239
3337	The Godfather	5893	8.4	7.851236
95	Interstellar	10867	8.1	7.809479
809	Forrest 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

We've built our initial (albeit basic) recommender system. Within these systems' 'Trending Now' section, we locate movies that are exceptionally popular, usually obtained by sorting the dataset based on the 'popularity' column.

In [10]:
```
pop= df2.sort_values('popularity', ascending=False)
import matplotlib.pyplot as plt
plt.figure(figsize=(12,4))
```

```
plt.barh(pop['title'].head(6),pop['popularity'].head(6), align='center',
        color='skyblue')
plt.gca().invert_yaxis()
plt.xlabel("Popularity")
plt.title("Popular Movies")
```

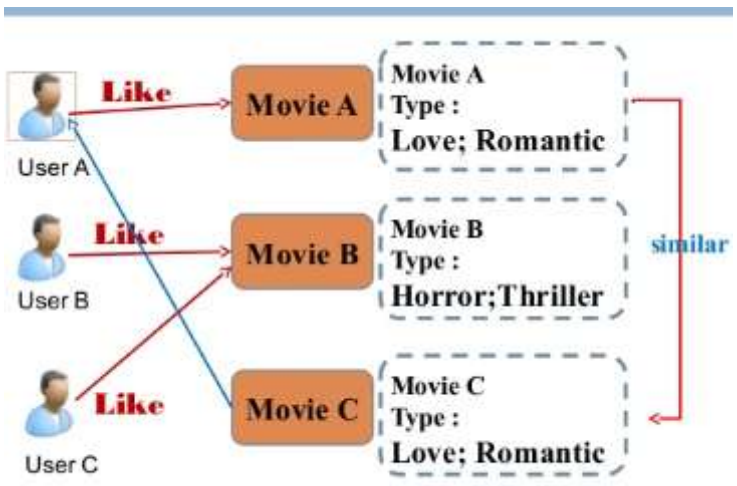
Out[10]: Text(0.5,1,'Popular Movies')



It's important to consider that demographic recommenders offer a broad chart of recommended movies to all users, lacking sensitivity to individual interests and tastes. This is where we transition to a more refined system - Content-Based Filtering.

Content Based Filtering

In this recommender system, the content of the movie (such as overview, cast, crew, keywords, tagline, etc.) is utilized to identify its similarity with other movies. Subsequently, movies that exhibit high similarity are recommended as they are most likely to align with a user's preferences.



Plot description based Recommender

We'll calculate pairwise similarity scores for all movies using their plot descriptions and make recommendations based on these scores. The plot descriptions are available in the 'overview' feature of our dataset. Let's dive into the data to get a closer look.

In [11]:
```
df2['overview'].head(5)
```

Out[11]:
```
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
```

For anyone who has dabbled in text processing, you'll know we need to convert the word vector of each overview. Hence, we'll compute Term Frequency-Inverse Document Frequency (TF-IDF) vectors for each overview.

If you're wondering about term frequency, it represents the relative frequency of a word in a document and is calculated as (term instances/total instances). Inverse Document Frequency, on the other hand, signifies the relative count of documents containing the term, given as log(number of documents/documents with term). The overall importance of each word to the documents in which they appear is determined by TF * IDF.

This process generates a matrix where each column represents a word in the overview vocabulary (comprising all words appearing in at least one document), and each row represents a movie, as before. This approach reduces the significance of words that frequently occur in plot overviews, impacting their weight in computing the final similarity score.

Fortunately, scikit-learn provides a built-in TfIdfVectorizer class that produces the TF-IDF matrix in just a few lines.

In [12]:
```
#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
```

Out[12]:
```
(4803, 20978)
```

We've observed the use of over 20,000 different words to describe the 4,800 movies in our dataset.

With this matrix ready, we can now compute a similarity score. There are several options for this, such as the Euclidean, Pearson, and cosine similarity scores. There isn't a definitive answer to which score is the best; different metrics perform well in different scenarios. Experimenting with various metrics is often a good approach.

In our case, we'll utilize the cosine similarity to compute a numeric value representing the similarity between two movies. The reason behind choosing cosine similarity is its independence from magnitude and its relatively easy and fast computation. Mathematically, it is defined as follows:

$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}},$$

Given our utilization of the TF-IDF vectorizer, the dot product directly provides the cosine similarity score. Hence, we'll opt for sklearn's linear_kernel() over cosine_similarities() for its faster computation.

In [13]:
```
# Import linear_kernel
from sklearn.metrics.pairwise import linear_kernel

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

We'll create a function that takes a movie title as input and generates a list of the 10 most similar movies. To achieve this, we require a reverse mapping of movie titles and DataFrame indices. Essentially, we need a mechanism to retrieve the index of a movie in our metadata DataFrame based on its title.

In [14]:
```
#Construct a reverse map of indices and movie titles
indices = pd.Series(df2.index, index=df2['title']).drop_duplicates()
```

Now, we're ready to define our recommendation function by following these steps:

Obtain the index of the movie given its title.

Retrieve the list of cosine similarity scores for that specific movie with all other movies and convert it into a list of tuples. Each tuple consists of the movie's position and its similarity score.

Sort the list of tuples based on the similarity scores (the second element).

Extract the top 10 elements from this sorted list, ignoring the first element as it refers to self (the movie most similar to itself).

Return the titles corresponding to the indices of the top elements.

In [15]:
```
# 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]
```

In [16]:
```
get_recommendations('The Dark Knight Rises')
```

Out[16]:
```
65                              The Dark Knight
299                              Batman Forever
428                              Batman Returns
1359                                    Batman
3854    Batman: The Dark Knight Returns, Part 2
119                               Batman Begins
2507                                  Slow Burn
9            Batman v Superman: Dawn of Justice
1181                                       JFK
210                              Batman & Robin
Name: title, dtype: object
```

In [17]:
```
get_recommendations('The Avengers')
```

Out[17]:
```
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
```

While our system successfully identifies movies with similar plot descriptions, the quality of recommendations isn't optimal. For instance, 'The Dark Knight Rises' retrieves all Batman movies, but it's more probable that individuals who enjoyed that movie are inclined to appreciate other Christopher Nolan movies. Unfortunately, our current system fails to capture this nuanced preference.

Credits, Genres and Keywords Based Recommender

It's evident that the quality of our recommender would significantly improve with the use of better metadata. That's precisely our aim in this section: building a recommender based on specific metadata—namely, the top 3 actors, director, related genres, and movie plot keywords.

To accomplish this, we'll extract the three most important actors, the director, and the keywords associated with each movie from the 'cast,' 'crew,' and 'keywords' features. Currently, our data is formatted as 'stringified' lists, requiring a conversion into a more secure and usable structure.

In [18]:
```
# Parse the stringified features into their corresponding python objects
from ast import literal_eval

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

Moving forward, we'll create functions designed to extract the necessary information from each feature.

In [19]:
```
# 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
```

In [20]:
```
# 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 []
```

In [21]:
```
# 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)
```

In [22]:
```
# Print the new features of the first 3 films
df2[['title', 'cast', 'director', 'keywords', 'genres']].head(3)
```

Out[22]:

	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]

Our subsequent step involves converting the names and keyword instances into lowercase while removing all spaces between them. This step ensures that our vectorizer doesn't treat 'Johnny' in 'Johnny Depp' and 'Johnny Galecki' as the same entity.

In [23]:
```
# 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 ''
```

In [24]:
```
# Apply clean_data function to your features.
features = ['cast', 'keywords', 'director', 'genres']

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

We're now ready to create our 'metadata soup,' a string containing all the desired metadata (such as actors, director, and keywords) that we'll feed into our vectorizer.

In [25]:
```
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 following steps mirror our approach with the plot description-based recommender. However, a key distinction lies in the utilization of CountVectorizer() instead of TF-IDF. This choice is deliberate—we aim to avoid down-weighting the presence of an actor or director based on their involvement in relatively more movies, which wouldn't align intuitively.

In [26]:
```
# 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'])
```

In [27]:
```
# 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)
```

In [28]:
```
# Reset index of our main DataFrame and construct reverse mapping as before
df2 = df2.reset_index()
indices = pd.Series(df2.index, index=df2['title'])
```

We're now able to reuse our 'get_recommendations()' function by passing in the new 'cosine_sim2' matrix as the second argument.

In [29]:
```
get_recommendations('The Dark Knight Rises', cosine_sim2)
```

Out[29]:
```
65               The Dark Knight
119                Batman Begins
4638    Amidst the Devil's Wings
1196                 The Prestige
3073           Romeo Is Bleeding
3326              Black November
1503                      Takers
1986                      Faster
303                     Catwoman
747                Gangster Squad
Name: title, dtype: object
```

In [30]:
```
get_recommendations('The Godfather', cosine_sim2)
```

Out[30]:
```
867      The Godfather: Part III
2731      The Godfather: Part II
4638    Amidst the Devil's Wings
2649           The Son of No One
1525              Apocalypse Now
1018             The Cotton Club
1170     The Talented Mr. Ripley
1209               The Rainmaker
1394               Donnie Brasco
1850                    Scarface
Name: title, dtype: object
```

Our recommender has effectively captured additional information with more metadata, resulting in arguably better recommendations. Fans of Marvel or DC comics, for instance, are more inclined to enjoy movies from the same production house. Hence, we can augment our existing features by including 'production_company.' Additionally, to enhance the director's significance, we can increase their weight in the 'soup' by adding the feature multiple times.

Collaborative Filtering

Our content-based engine faces severe limitations. It can only suggest movies similar to a specific movie, lacking the capability to encompass diverse tastes and offer cross-genre recommendations.

Additionally, the engine we constructed lacks personalization as it doesn't consider individual tastes and biases. Anyone querying our engine for movie recommendations based on a specific film will receive identical suggestions, irrespective of their preferences.

Hence, in this section, we'll adopt a technique known as Collaborative Filtering to provide recommendations for Movie Watchers. This technique primarily comprises two types:

- **User based filtering**- These systems suggest products based on what similar users have liked. To measure similarity between two users, we can utilize either Pearson correlation or cosine similarity. This filtering technique becomes clearer with an example. In the following matrices, each row represents a user, and the columns correspond to various movies, except for the last column which denotes the similarity between that user and the target user. Each cell reflects the rating given by the user to that specific movie. Let's assume that user E is the target.

	The Avengers	Sherlock	Transformers	Matrix	Titanic	Me Before You	Similarity(i, E)
A	2		2	4	5		NA
B	5		4			1	
C			5		2		
D		1		5		4	
E			4			2	1
F	4	5		1			NA

Given that users A and F do not have any shared movie ratings with user E, their similarities with user E cannot be defined using Pearson Correlation. Hence, our consideration is solely directed towards users B, C, and D. Using Pearson Correlation, we can calculate the following similarities.

	The Avengers	Sherlock	Transformers	Matrix	Titanic	Me Before You	Similarity(i, E)
A	2		2	4	5		NA
B	5		4			1	0.87
C			5		2		1
D		1		5		4	-1
E			4			2	1
F	4	5		1			NA

The table above illustrates that user D significantly differs from user E, evident from their negative Pearson Correlation. This contrast arises as user D rated 'Me Before You' higher than their average rating, while user E did the opposite. Now, we can proceed to infer the missing ratings for the movies that user E hasn't rated based on other users.

	The Avengers	Sherlock	Transformers	Matrix	Titanic	Me Before You	Similarity(i, E)
A	2		2	4	5		NA
B	5		4			1	0.87
C			5		2		1
D		1		5		4	-1
E	3.51*	3.81*	4	2.42*	2.48*	2	1
F	4	5		1			NA

While computing user-based Collaborative Filtering is straightforward, it faces several challenges. One significant issue is the potential change in users' preferences over time. This suggests that precomputing the matrix based on their neighboring users might result in poor performance. To address this concern, we can implement item-based Collaborative Filtering.

- **Item Based Collaborative Filtering** - Unlike measuring the similarity between users, item-based Collaborative Filtering recommends items based on their similarity to the items rated by the target user. Similar to user-based CF, similarity computation can be conducted using Pearson Correlation or Cosine Similarity. However, the key distinction lies in the process: with item-based Collaborative Filtering, we fill in the blank vertically, as opposed to the horizontal manner characteristic of user-based CF. The

following table illustrates this process for the movie 'Me Before You'.

	The Avengers	Sherlock	Transformers	Matrix	Titanic	Me Before You
A	2		2	4	5	2.94*
B	5		4			1
C			5		2	2.48*
D		1		5		4
E			4			2
F	4	5		1		1.12*
Similarity	-1	-1	0.86	1	1	

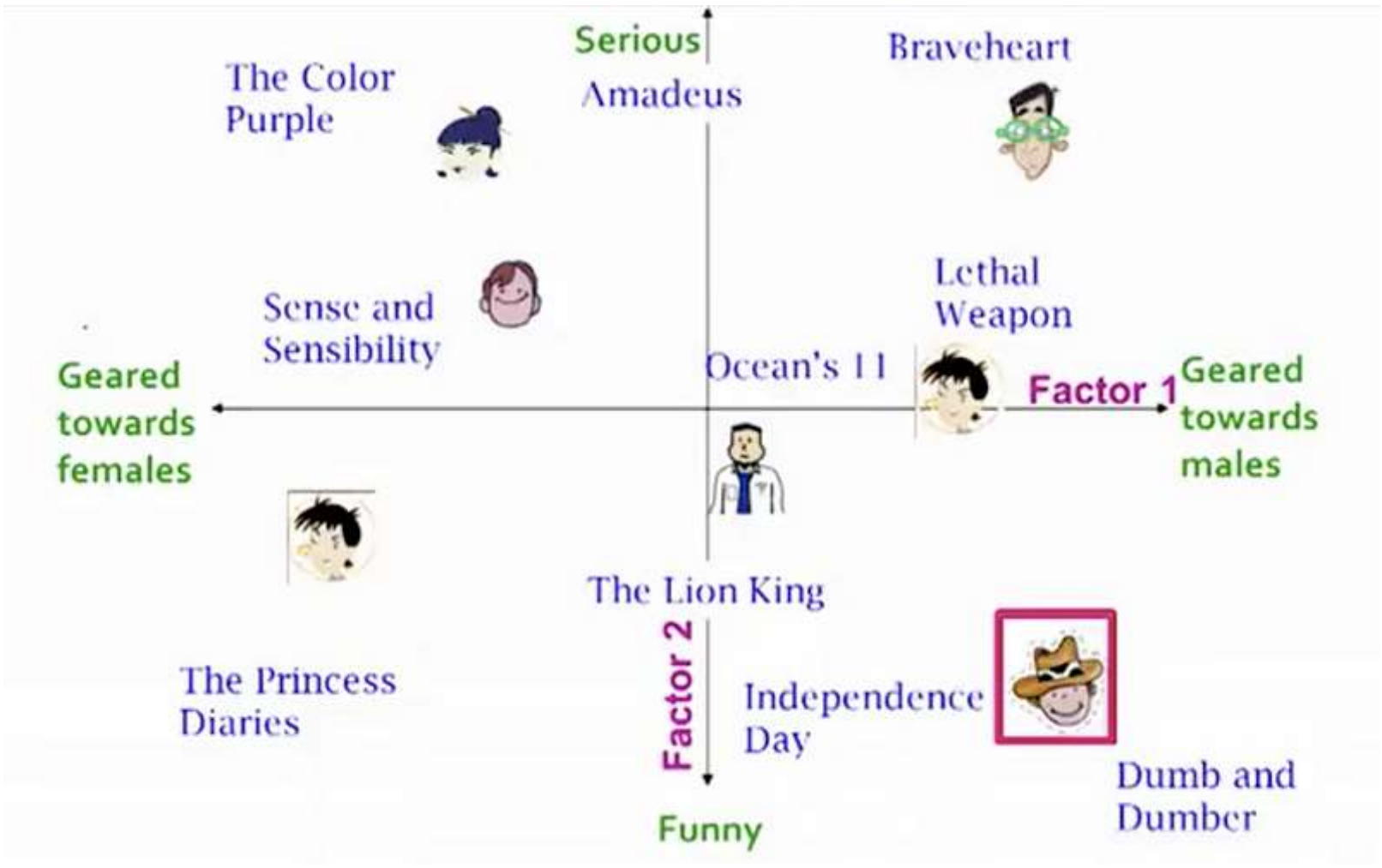
While item-based Collaborative Filtering addresses the issue of dynamic user preferences by being more static, several problems persist within this method. Primarily, scalability becomes a major concern. The computational demands increase proportionally with both the number of customers and the products. In the worst-case scenario, the complexity can grow to O(mn) with m users and n items.

Furthermore, sparsity poses another challenge. Looking at the earlier table, the similarity between 'Matrix' and 'Titanic' is 1, based on just one user who rated both movies. In extreme cases involving millions of users, two fairly different movies might exhibit very high similarity simply because they have a similar rank from the single user who rated them both.

Single Value Decomposition

One approach to address the scalability and sparsity issues inherent in Collaborative Filtering is to employ a latent factor model, which captures the similarity between users and items. Essentially, this approach transforms the recommendation problem into an optimization problem, aiming to predict ratings for items given a user. One commonly used metric for evaluating such predictions is Root Mean Square Error (RMSE), where lower RMSE indicates better performance.

Now, you might wonder: what exactly is a latent factor? It's a broad concept representing a property or concept associated with a user or an item. For example, in music, a latent factor might correspond to the genre that the music belongs to. Singular Value Decomposition (SVD) reduces the dimensionality of the utility matrix by extracting its latent factors. This process essentially maps each user and item into a latent space of dimension 'r.' Consequently, it facilitates a better understanding of the relationship between users and items, making them directly comparable. The figure below illustrates this concept.



let's delve into implementation. As our previous dataset lacked userId (essential for collaborative filtering), we'll load another dataset. To implement Singular Value Decomposition (SVD), we'll use the Surprise library.

In [31]:
```
from surprise import Reader, Dataset, SVD, evaluate
reader = Reader()
ratings = pd.read_csv('../input/the-movies-dataset/ratings_small.csv')
ratings.head()
```

Out[31]:

	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.

In [32]:
```
data = Dataset.load_from_df(ratings[['userId', 'movieId', 'rating']], reader)
data.split(n_folds=5)
```

In [33]:
```
svd = SVD()
evaluate(svd, data, measures=['RMSE', 'MAE'])
```

```
/opt/conda/lib/python3.6/site-packages/surprise/evaluate.py:66: UserWarning: The evaluate() method is deprecated. Pleas
e use model_selection.cross_validate() instead.
  'model_selection.cross_validate() instead.', UserWarning)
/opt/conda/lib/python3.6/site-packages/surprise/dataset.py:193: UserWarning: Using data.split() or using load_from_fold
s() without using a CV iterator is now deprecated.
  UserWarning)
Evaluating RMSE, MAE of algorithm SVD.

------------
Fold 1
RMSE: 0.8960
MAE:  0.6914
------------
Fold 2
RMSE: 0.8981
MAE:  0.6910
------------
Fold 3
RMSE: 0.8938
MAE:  0.6884
------------
Fold 4
RMSE: 0.9039
MAE:  0.6949
------------
Fold 5
RMSE: 0.8959
MAE:  0.6912
------------
------------
Mean RMSE: 0.8975
Mean MAE : 0.6914
------------
------------
```

Out[33]:
```
CaseInsensitiveDefaultDict(list,
                           {'rmse': [0.8960391214492476,
                             0.8980704187280663,
                             0.8937535450114095,
                             0.9038766692981838,
                             0.8958994353207446],
                            'mae': [0.691353530022487,
                             0.6910441809693095,
                             0.688386398499374,
                             0.6948636174486761,
                             0.6912240802634083]})
```

We achieve a mean Root Mean Square Error of approximately 0.89, which is more than satisfactory for our requirements. Now, let's proceed to train on our dataset and generate predictions.

In [34]:
```
trainset = data.build_full_trainset()
svd.fit(trainset)
```

Out[34]:
```
<surprise.prediction_algorithms.matrix_factorization.SVD at 0x7c4e817c1780>
```

Let's select the user with User ID 1 and examine the ratings they have given.

In [35]:
```
ratings[ratings['userId'] == 1]
```

Out[35]:

	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
7	1	1293	2.0	1260759148
8	1	1339	3.5	1260759125
9	1	1343	2.0	1260759131
10	1	1371	2.5	1260759135
11	1	1405	1.0	1260759203
12	1	1953	4.0	1260759191
13	1	2105	4.0	1260759139
14	1	2150	3.0	1260759194
15	1	2193	2.0	1260759198
16	1	2294	2.0	1260759108
17	1	2455	2.5	1260759113
18	1	2968	1.0	1260759200
19	1	3671	3.0	1260759117

In [36]:
```
svd.predict(1, 302, 3)
```

Out[36]:
```
Prediction(uid=1, iid=302, r_ui=3, est=2.463188184288999, details={'was_impossible': False})
```

Considering the movie with ID 302, our estimated prediction stands at 2.618. One notable characteristic of this recommender system is its indifference towards the content or nature of the movie itself. It operates solely on the assigned movie ID, predicting ratings based on how other users have rated the movie.

Conclusion

We create recommenders using demographic , content- based and collaborative filtering. While demographic filtering is very elemantary and cannot be used practically, **HWe've developed recommenders employing demographic, content-based, and collaborative filtering. While demographic filtering is basic and impractical, Hybrid Systems can capitalize on the complementary nature of content-based and collaborative filtering. This model serves as a basic starting framework, providing fundamental insights.ybrid Systems** can take advantage of content-based and collaborative filtering as the two approaches are proved to be almost complimentary.

This model was very baseline and only provides a fundamental framework to start with.