Recommendation Systems

Companies like Amazon(books, items), Netflix(movies), Google(News,Search), and Pandora/Spotify(music) leverage recommendation systems to help users discover new and relevant items (products, videos, jobs, music), creating a delightful user experience while driving incremental revenue.

The need to build robust recommendation systems is extremely important given the huge demand for personalized content of modern consumers.

In this assignment, you will be applying your learning of recommendation systems in this Unit towards building the following four different types of recommendation systems:

1. Global Recommendation Systems (Statistical)
2. Content-based Recommendation Systems
3. Collaborative Filtering (User-Item) Recommendation Systems
4. Hybrid Recommendation Systems

The focus of the mini-project here would be to build a movie recommendation system.

1. Dataset Acquisition

Following are the key descriptions of the datasets you will be using. The data used here has been compiled from various movie datasets like Netflix and IMDb.

1. **Filename: `movie_titles.csv` :**

    - **`MovieID`** : MovieID does not correspond to actual Netflix movie ids or IMDB movie ids
    - **`YearOfRelease`** : YearOfRelease can range from 1890 to 2005 and may correspond to the release of corresponding DVD, not necessarily its theaterical release
    - **`Title`** : Title is the Netflix movie title and may not correspond to titles used on other sites. Titles are in English

1. **Combined User-Ratings Dataset Description - `combined_data.csv` :**

    - The first line of the contains the movie id followed by a colon.
    - Each subsequent line in the file corresponds to a rating from a customer and its date in the following format:
        - MovieIDs range from 1 to 17770 sequentially.
        - CustomerIDs range from 1 to 2649429, with gaps. There are 480189 users.
        - Ratings are on a five star (integral) scale from 1 to 5.
        - Dates have the format YYYY-MM-DD.

1. **Filename: `movies_metadata.csv`**

The main Movies Metadata file. Contains information on 45,000 movies featured in the Full MovieLens dataset. Features include posters, backdrops, budget, revenue, release dates, languages, production countries and companies.

2: Import Necessary Dependencies

We will be leveraging **`keras`** on top of **`tensorflow`** for building some of the collaborative filtering and hybrid models. There are compatibility issues with handling sparse layers with dense layers till now in TensorFlow 2 hence we are leveraging native Keras but in the long run once this issue is resolved we can leverage **`tf.keras`** with minimal code updates.

In [1]:

```
# filter out unncessary warnings
import warnings
warnings.filterwarnings('ignore')
```

In [68]:

```
# To store\load the data
import pandas as pd

# To do linear algebra
import numpy as np

# To create plots
import matplotlib.pyplot as plt
import seaborn as sns


# To compute similarities between vectors
from sklearn.metrics import mean_squared_error
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.feature_extraction.text import TfidfVectorizer

# data load progress bars
from tqdm import tqdm

from collections import deque

# To create deep learning models
import tensorflow as tf
import keras
from keras.layers import Input, Embedding, Reshape, Dot, Concatenate, Dense, Dropout
from keras.models import Model

# To stack sparse matrices
from scipy.sparse import vstack
```

In [69]:

```
# remove unnecessary TF logs
import logging
tf.get_logger().setLevel(logging.ERROR)
```

In [70]:

```
# check keras and TF version used
print('TF Version:', tf.__version__)
print('Keras Version:', keras.__version__)
```

```
# TF Version: 1.15.0
# Keras Version: 2.2.5
```

```
TF Version: 1.15.0
Keras Version: 2.2.5
```

Let's start loading data that will be used for building the recommendation systems

3. Load Datasets

3.1: Load Movie Metadata Datasets

First, we will load the movie_titles.csv data from the Netflix prize data source

In [5]:

```
# Load data for all movies
movie_titles = pd.read_csv('./data/movie_titles.csv.zip',
                           encoding = 'ISO-8859-1',
                           header = None,
                           names = ['Id', 'Year', 'Name']).set_index('Id')

print('Shape Movie-Titles:\t{}'.format(movie_titles.shape))
movie_titles.sample(5)
```

```
Shape Movie-Titles:	(17770, 2)
```

Out[5]:

Id	Year	Name
10832	1960.0	Psycho
15514	2002.0	Behind the Red Door
508	1979.0	Saint Jack
15608	1972.0	The Bitter Tears of Petra Von Kant
13903	2003.0	National Geographic: Inside Special Forces

There are approximately 18000 movies in the ratings dataset and the metadata information includes the year of release and movie title

Next, we will load the movie_metadata.csv from The movies dataset source. This is to get the metadata information like description etc. related to each movie.

In [6]:

```
# Load a movie metadata dataset
movie_metadata = (pd.read_csv('./data/movies_metadata.csv.zip',
                              low_memory=False)[['original_title', 'overview', 'vote_co
                  .set_index('original_title')
                  .dropna())

# Remove the long tail of rarly rated moves
movie_metadata = movie_metadata[movie_metadata['vote_count']>10].drop('vote_count', axi

print('Shape Movie-Metadata:\t{}'.format(movie_metadata.shape))
movie_metadata.sample(5)
```

```
Shape Movie-Metadata:   (21604, 1)
```

Out[6]:

original_title	overview
Nick Offerman: American Ham	This live taping of Nick Offerman's hilarious ...
The Magical Legend of the Leprechauns	American businessman Jack Woods rents a cottag...
Jeremiah Johnson	A mountain man who wishes to live the life of ...
Le Nouveau	Benoit is the new kid at a junior high school....
Deux jours à tuer	Antoine Méliot is around 40 years old and has ...

Around 21,000 entries in the movies metadata dataset

3.2: Load User-Movie-Rating Dataset

In [7]:

```
# Dowload large file from the shared GDrive folder
#!pip install gdown
#!gdown "https://drive.google.com/uc?export=download&id=1z0O0fXuofdsbpL8fkCVgjeIwFP_LxG
```

In [8]:

```
# Load single data-file
df_raw = pd.read_csv('./data/combined_data.csv.zip',
                     header=None,
                     names=['User', 'Rating', 'Date'],
                     usecols=[0, 1, 2])

# Find empty rows to slice dataframe for each movie
tmp_movies = df_raw[df_raw['Rating'].isna()]['User'].reset_index()
movie_indices = [[index, int(movie[:-1])] for index, movie in tmp_movies.values]

# Shift the movie_indices by one to get start and endpoints of all movies
shifted_movie_indices = deque(movie_indices)
shifted_movie_indices.rotate(-1)

# Gather all dataframes
user_data = []

# Iterate over all movies
for [df_id_1, movie_id], [df_id_2, next_movie_id] in zip(movie_indices, shifted_movie_i

    # Check if it is the last movie in the file
    if df_id_1<df_id_2:
        tmp_df = df_raw.loc[df_id_1+1:df_id_2-1].copy()
    else:
        tmp_df = df_raw.loc[df_id_1+1:].copy()

    # Create movie_id column
    tmp_df['Movie'] = movie_id

    # Append dataframe to list
    user_data.append(tmp_df)

# Combine all dataframes
df = pd.concat(user_data)
del user_data, df_raw, tmp_movies, tmp_df, shifted_movie_indices, movie_indices, df_id_
```

```
print('Shape User-Ratings:\t{}'.format(df.shape))
df.sample(10)
```

```
Shape User-Ratings:	(24053764, 4)
```

Out[8]:

	User	Rating	Date	Movie
23455386	2543612	3.0	2005-07-17	4389
12160237	2630793	4.0	2004-10-04	2360
2052944	1971069	4.0	2004-11-22	375
12257468	2225873	5.0	2005-01-27	2372
23905462	1991006	2.0	2004-05-04	4472
3354700	1208512	5.0	2004-09-02	629
2700036	86224	3.0	2005-08-04	483
20730549	54813	4.0	2005-03-15	3917
23520417	754909	3.0	2001-10-18	4393
13468273	1299574	4.0	2005-07-18	2578

There are about 24 Million+ different rating records!

We have taken the data required for building the system and now let's do some EDA on the dataset to better understand our data

4. Exploratory Data Analysis

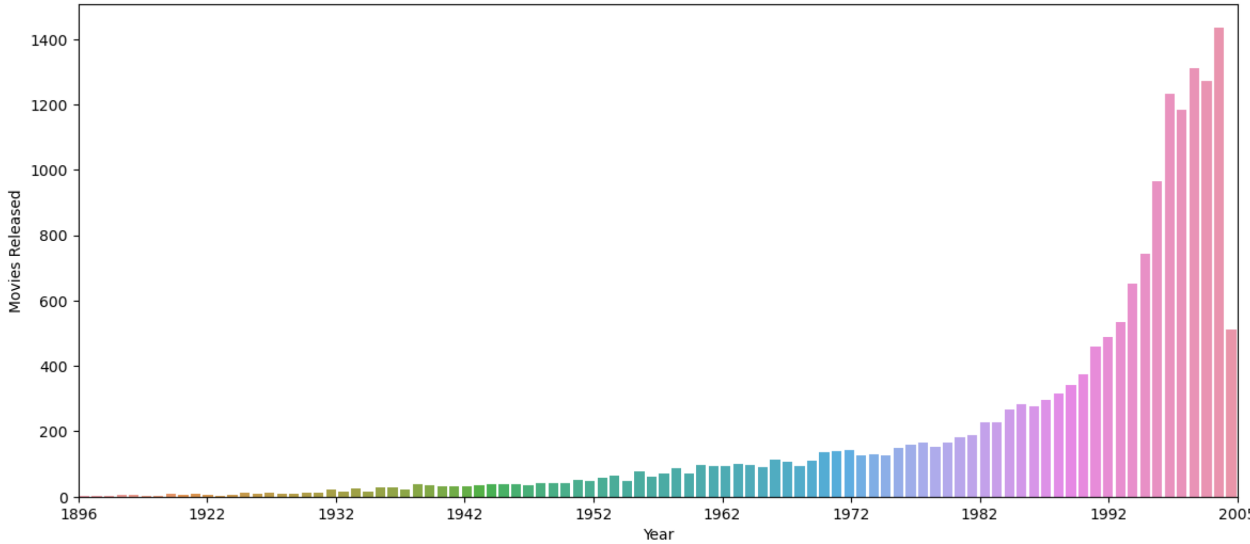
4.1: When were the movies released?

In [75]:

```
fig, ax = plt.subplots(1, 1, figsize=(14, 6))

data = movie_titles['Year'].value_counts().sort_index()
x = data.index.map(int)
y = data.values

sns.barplot(x, y)
xmin, xmax = plt.xlim()
xtick_labels = [x[0]] + list(x[10:-10:10]) + [x[-1]]
plt.xticks(ticks=np.linspace(xmin, xmax, 10), labels=xtick_labels);
ax.set_xlabel('Year')
ax.set_ylabel('Movies Released')
plt.show()
```



Many movies on Netflix have been released in this millennial. Whether Netflix prefers young movies or there are no old movies left can not be deduced from this plot. The decline for the rightmost point is probably caused by an incomplete last year.

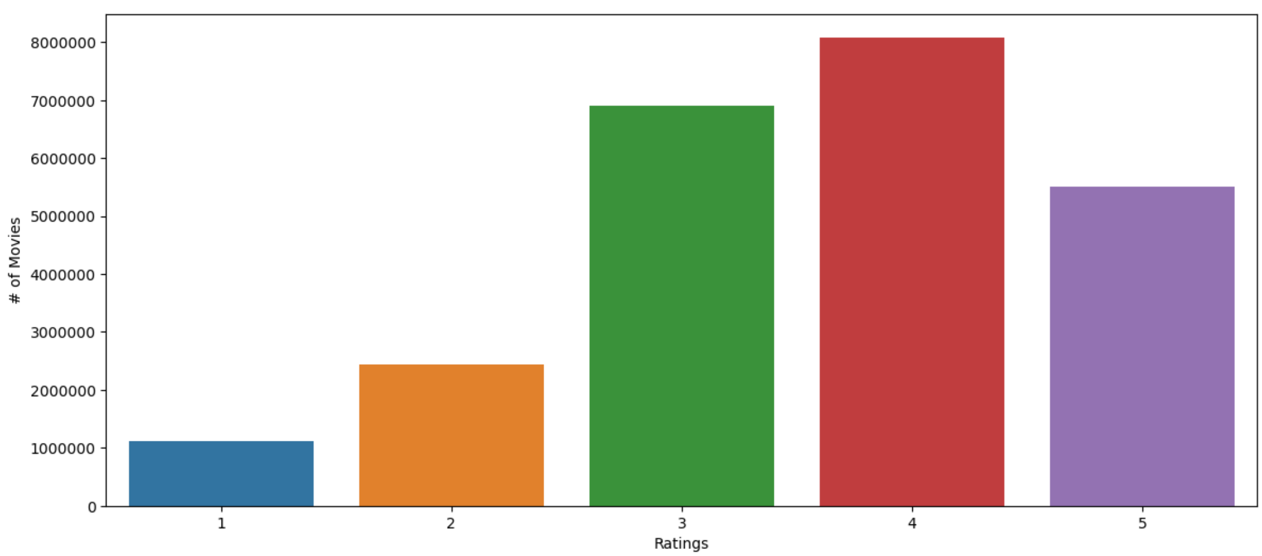
Q 4.2: How are The Ratings Distributed?

Your Turn: Build the visualization for rating distributions similar to the previous plot.

In [76]:
```
fig, ax = plt.subplots(1, 1, figsize=(14, 6))

data = df['Rating'].value_counts().sort_index()
x = data.index.map(int)
y = data.values

sns.barplot(x, y)
plt.ticklabel_format(style='plain', axis='y',useOffset=False)
ax.set_xlabel('Ratings')
ax.set_ylabel('# of Movies')
plt.show()
```



Netflix movies rarely have a rating lower than three. Most ratings have between three and four stars.

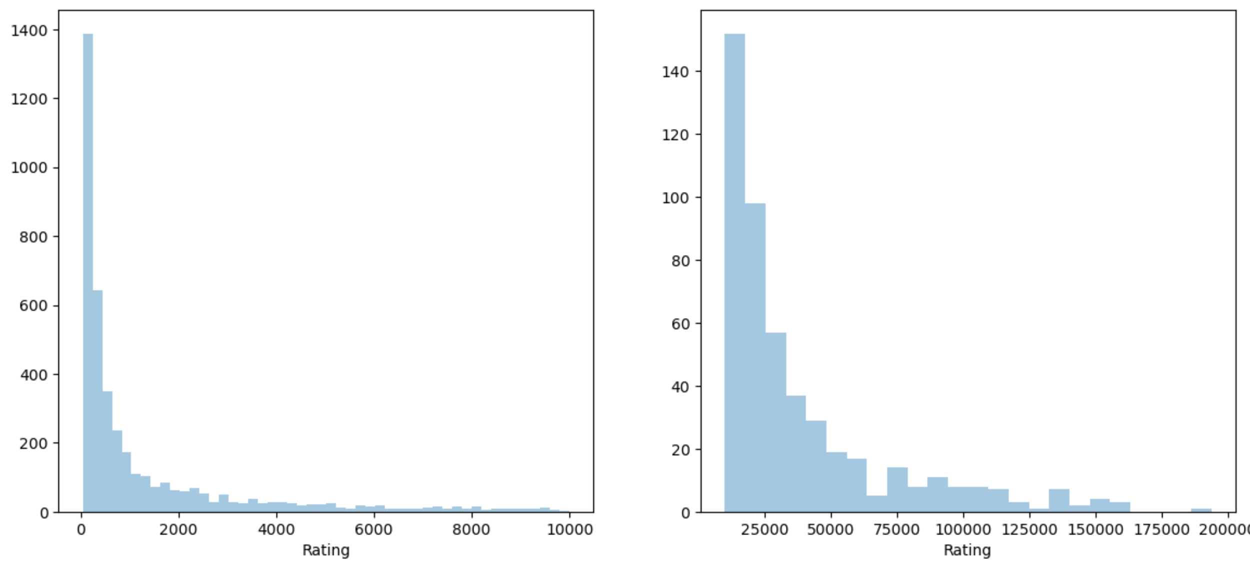
The distribution is probably biased, since only people liking the movies proceed to be customers and others presumably will leave the platform.

4.3: Visualize the Distribution of Number of Movie Ratings

This is to understand how many movies (y-axis) are receiving specific number of movie ratings (x-axis)

In [79]:
```
fig, ax = plt.subplots(1, 2, figsize=(14, 6))

data = df.groupby('Movie')['Rating'].count()
sns.distplot(data[data  < 10000], kde=False, ax=ax[0]);
sns.distplot(data[data  > 10000], kde=False, ax=ax[1]);
plt.show()
```



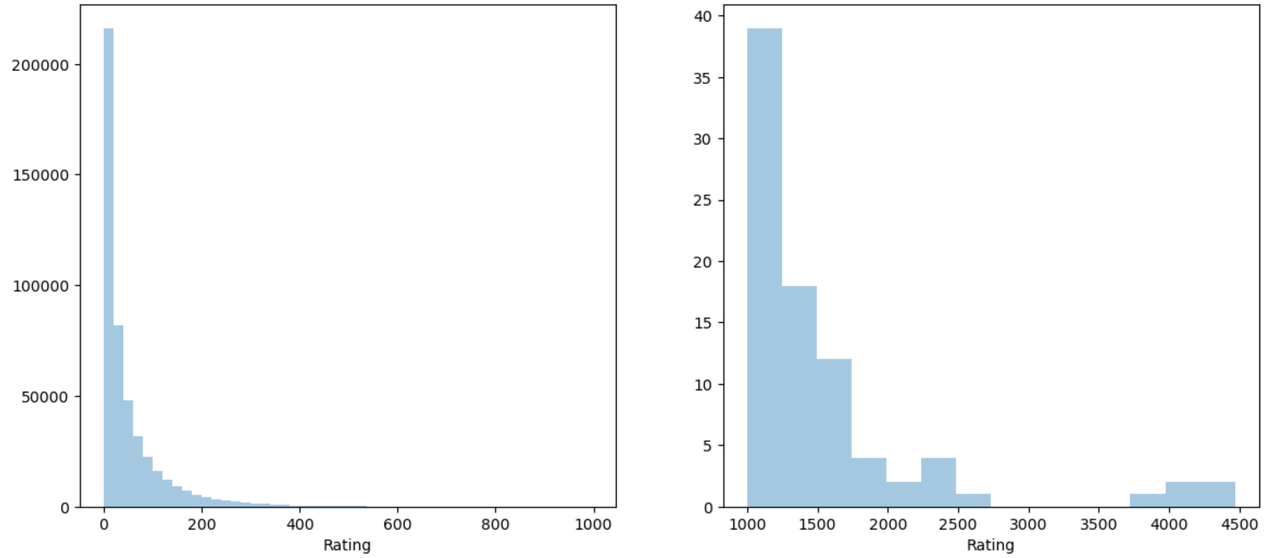
Q 4.4: Visualize the Distribution of Number of User Ratings

This is to understand how many users (y-axis) are giving specific number of movie ratings (x-axis)

Your Turn: Try to find out an optimal threshold as in the previous example to split the data to form two understandable subplots!

In [80]:
```
fig, ax = plt.subplots(1, 2, figsize=(14, 6))

data = df.groupby('User')['Rating'].count()
sns.distplot(data[data  < 1000], kde=False, ax=ax[0]);
sns.distplot(data[data  > 1000], kde=False, ax=ax[1]);
plt.show()
```



The ratings per movie as well as the ratings per user both have nearly a perfect exponential decay. Only very few movies/users have many ratings.

5. Dimensionality Reduction & Filtering

Filter Sparse Movies And Users

To reduce the dimensionality of the dataset I am filtering rarely rated movies and rarely rating users out.

In [13]:

```
# Filter sparse movies
min_movie_ratings = 1000
filter_movies = (df['Movie'].value_counts()>min_movie_ratings)
filter_movies = filter_movies[filter_movies].index.tolist()

# Filter sparse users
min_user_ratings = 200
filter_users = (df['User'].value_counts()>min_user_ratings)
filter_users = filter_users[filter_users].index.tolist()

# Actual filtering
df_filtered = df[(df['Movie'].isin(filter_movies)) & (df['User'].isin(filter_users))]
del filter_movies, filter_users, min_movie_ratings, min_user_ratings
print('Shape User-Ratings unfiltered:\t{}'.format(df.shape))
print('Shape User-Ratings filtered:\t{}'.format(df_filtered.shape))
```

```
Shape User-Ratings unfiltered:  (24053764, 4)
Shape User-Ratings filtered:    (5930581, 4)
```

After filtering sparse movies and users about 5.9M rating records are present.

6. Create Train and Test Datasets

Do note this will be used for the statistical method based models and collaborative filtering.

For content based filtering it is more of a model which recommends movies rather than predicting ratings and for the hybrid model we will need to recreate the train and test datasets later since we need to create a subset of movies-users-ratings which have movie text descriptions.

Create Train and Test datasets

In [14]:
```
# Shuffle DataFrame
df_filtered = df_filtered.drop('Date', axis=1).sample(frac=1).reset_index(drop=True)

# Testingsize
n = 100000

# Split train- & testset
df_train = df_filtered[:-n]
df_test = df_filtered[-n:]
df_train.shape, df_test.shape
```

Out[14]:
```
((5830581, 3), (100000, 3))
```

The train set will be used to train all models and the test set ensures we can compare model performance on unseen data using the RMSE metric.

7. Transformation

Q 7.1: Transform The User-Movie-Ratings Data Frame to User-Movie Matrix

A large, sparse matrix will be created in this step. Each row will represent a user and its ratings and the columns are the movies.

The movies already rated by users are the non-empty values in the matrix.

Empty values are unrated movies and the main objective is to estimate the empty values to help our users.

Your turn: Create the User-Movie matrix leveraging the `pivot_table()` function from pandas.

Fill in the blanks in the code below by referencing the `pivot_table()` function and invoking it on `df_train`. Feel free to check out the documentation.

Remember, rows should be users, columns should be movies and the values in the matrix should be the movie ratings. All these should be available in the `df_train` dataframe.

In [15]:
```
# Create a user-movie matrix with empty values
df_p = pd.pivot_table(df_train, values='Rating', index=['User'], columns=['Movie'] )
print('Shape User-Movie-Matrix:\t{}'.format(df_p.shape))
df_p.head(10)
```

```
Shape User-Movie-Matrix:	(20828, 1741)
```

Out[15]:

Movie	3	5	6	8	16	17	18	24	25	26	...	4482	4483	4484	4485	44

Movie	3	5	6	8	16	17	18	24	25	26	...	4482	4483	4484	4485	44
User																
1000079	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	2
1000192	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	Na
1000301	NaN	NaN	NaN	NaN	NaN	NaN	4.0	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	4
1000387	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	1.0	2
1000410	NaN	NaN	NaN	NaN	NaN	NaN	4.0	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	3
1000527	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	Na
1000596	NaN	NaN	NaN	NaN	NaN	NaN	NaN	2.0	NaN	NaN	...	NaN	NaN	NaN	NaN	Na
1000634	NaN	NaN	NaN	NaN	3.0	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	4
1000710	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	Na
1000779	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	Na

10 rows × 1741 columns

8. Building Recommendation Systems

8.1(a): Global Recommendation Systems (Mean Rating)

Computing the mean rating for all movies creates a ranking. The recommendation will be the same for all users and can be used if there is no information on the user. Variations of this approach can be separate rankings for each country/year/gender/... and to use them individually to recommend movies/items to the user.

It has to be noted that this approach is biased and favours movies with fewer ratings, since large numbers of ratings tend to be less extreme in its mean ratings.

Additional Hint

Predict model performance: mean_squared_error

In [16]:

```
# Compute mean rating for all movies
ratings_mean = df_p.mean(axis=0).sort_values(ascending=False).rename('Rating-Mean').to_

# Compute rating frequencies for all movies
ratings_count = df_p.count(axis=0).rename('Rating-Freq').to_frame()

# Combine the aggregated dataframes
combined_df = ratings_mean.join(ratings_count).join(movie_titles)
combined_df.head(5)
```

Out[16]:

	Rating-Mean	Rating-Freq	Year	Name

Movie	Rating-Mean	Rating-Freq	Year	Name
Movie				
3456	4.655775	1316	2004.0	Lost: Season 1
2102	4.504488	2785	1994.0	The Simpsons: Season 6
3444	4.435798	2827	2004.0	Family Guy: Freakin' Sweet Collection
2452	4.425515	18601	2001.0	Lord of the Rings: The Fellowship of the Ring
2172	4.387149	6194	1991.0	The Simpsons: Season 3

In [17]:

```
# Join labels and predictions based on mean movie rating
predictions_df = df_test.set_index('Movie').join(ratings_mean)
predictions_df.head(5)
```

Out[17]:

	User	Rating	Rating-Mean
Movie			
3	2440601	4.0	3.441065
3	2141037	5.0	3.441065
3	25049	4.0	3.441065
3	971925	5.0	3.441065
3	2113709	4.0	3.441065

In [18]:

```
# Compute RMSE
y_true = predictions_df['Rating']
y_pred = predictions_df['Rating-Mean']

rmse = np.sqrt(mean_squared_error(y_true=y_true, y_pred=y_pred))
print("The RMSE Value for the Mean Rating Recommender:", rmse)
```

```
The RMSE Value for the Mean Rating Recommender: 1.0076583914488069
```

In [19]:

```
# View top ten rated movies
combined_df[['Name', 'Rating-Mean']].head(10)
```

Out[19]:

	Name	Rating-Mean
Movie		
3456	Lost: Season 1	4.655775
2102	The Simpsons: Season 6	4.504488
3444	Family Guy: Freakin' Sweet Collection	4.435798
2452	Lord of the Rings: The Fellowship of the Ring	4.425515
2172	The Simpsons: Season 3	4.387149
1256	The Best of Friends: Vol. 4	4.375553

	Name	Rating-Mean
Movie		
3962	Finding Nemo (Widescreen)	4.368685
4238	Inu-Yasha	4.354839
3046	The Simpsons: Treehouse of Horror	4.349623
1476	Six Feet Under: Season 4	4.347809

Q 8.1(b): Global Recommendation Systems (Weighted Rating)

To tackle the problem of the unstable mean with few ratings e.g. IDMb uses a weighted rating. Many good ratings outweigh few in this algorithm.

Hint:

Weighted Rating Formula

weighted rating $(WR) = (v/(v+m))R + (m/(v+m))C$

where:

R = average for the movie (mean) = (Rating)

v = number of votes for the movie = (votes)

m = minimum votes required

C = the mean vote across the whole report

Your Turn: Fill in the necessary code snippets below to build and test the model

In [20]:
```
# Number of minimum votes to be considered
m = 1000

# Mean rating for all movies
C = df_p.stack().mean()

# Mean rating for all movies separately
R = df_p.mean(axis=0).values

# Rating freqency for all movies separately
v = df_p.count().values
```

In [21]:
```
# Weighted formula to compute the weighted rating
weighted_score = (v / (v+m)) * R + (m / (v+m)) * C
```

In [22]:
```
# convert weighted_score into a dataframe
```

```
weighted_mean = pd.Series(data=weighted_score, index=df_p.mean(axis=0).index).rename("R

# Combine the aggregated dataframes (wighted_mean & movie_titles)
combined_df = weighted_mean.join(movie_titles)
combined_df.head(5)
```

Out[22]:

Movie	Rating-WM	Year	Name
3	3.458672	1997.0	Character
5	3.454992	2004.0	The Rise and Fall of ECW
6	3.375730	1997.0	Sick
8	3.165109	2004.0	What the #$*! Do We Know!?
16	3.197685	1996.0	Screamers

In [23]:

```
# Join labels and predictions based on mean movie rating
predictions_df = df_test.set_index('Movie').join(weighted_mean)
predictions_df.head(5)
```

Out[23]:

Movie	User	Rating	Rating-WM
3	2440601	4.0	3.458672
3	2141037	5.0	3.458672
3	25049	4.0	3.458672
3	971925	5.0	3.458672
3	2113709	4.0	3.458672

In [24]:

```
# Compute RMSE
y_true = predictions_df['Rating']
y_pred = predictions_df['Rating-WM']

rmse = np.sqrt(mean_squared_error(y_true=y_true, y_pred=y_pred))
print("The RMSE Value for the Weighted-Mean Rating Recommender:", rmse)
```

```
The RMSE Value for the Weighted-Mean Rating Recommender: 1.0126471751297588
```

In [25]:

```
# View top ten rated movies
combined_df.sort_values(by=["Rating-WM"], ascending=False).head(10)
```

Out[25]:

Movie	Rating-WM	Year	Name
2452	4.376661	2001.0	Lord of the Rings: The Fellowship of the Ring
3962	4.320046	2003.0	Finding Nemo (Widescreen)
4306	4.289505	1999.0	The Sixth Sense

	Rating-WM	Year	Name
Movie			
2862	4.283415	1991.0	The Silence of the Lambs
3290	4.264066	1974.0	The Godfather
2172	4.259374	1991.0	The Simpsons: Season 3
2102	4.230630	1994.0	The Simpsons: Season 6
2782	4.216726	1995.0	Braveheart
3046	4.203503	1990.0	The Simpsons: Treehouse of Horror
3864	4.184219	2005.0	Batman Begins

The variable "m" can be seen as regularizing parameter. Changing it determines how much weight is put onto the movies with many ratings. Even if there is a better ranking the RMSE decreased slightly. There is a trade-off between interpretability and predictive power.

8.2: Content Based Recommendation Systems

The Content-Based Recommender relies on the similarity of the items being recommended. The basic idea is that if you like an item, then you will also like a “similar” item. It generally works well when it’s easy to determine the context/properties of each item. If there is no historical data for a user or there is reliable metadata for each movie, it can be useful to compare the metadata of the movies to find similar ones.

Cosine TFIDF Movie Description Similarity

TF-IDF

This is a text vectorization technique which is used to determine the relative importance of a document / article / news item / movie etc.

TF is simply the frequency of a word in a document.

IDF is the inverse of the document frequency among the whole corpus of documents.

TF-IDF is used mainly because of two reasons: Suppose we search for “the results of latest European Socccer games” on Google. It is certain that “the” will occur more frequently than “soccer games” but the relative importance of soccer games is higher than the search query point of view.

In such cases, TF-IDF weighting negates the effect of high frequency words in determining the importance of an item (document).

Cosine Similarity

After calculating TF-IDF scores, how do we determine which items are closer to each other, rather closer to the user profile? This is accomplished using the Vector Space Model which computes the proximity based on the angle between the vectors.

Consider the following example

Sentence 2 is more likely to be using Term 2 than using Term 1. Vice-versa for Sentence 1.

The method of calculating this relative measure is calculated by taking the cosine of the angle between the sentences and the terms.

The ultimate reason behind using cosine is that the value of cosine will increase with decreasing value of the angle between which signifies more similarity.

The vectors are length normalized after which they become vectors of length 1 and then the cosine calculation is simply the sum-product of vectors.

In this approch we will use the movie description to create a TFIDF-matrix, which counts and weights words in all descriptions, and compute a cosine similarity between all of those sparse text-vectors. This can easily be extended to more or different features if you like. It is impossible for this model to compute a RMSE score, since the model does not recommend the movies directly. In this way it is possible to find movies closly related to each other.

This approach of content based filtering can be extendend to increase the model performance by adding some more features like genres, cast, crew etc.

In [26]:

```
# view sample movie descriptions
movie_metadata['overview'].head(5)
```

Out[26]:

```
original_title
Toy Story                      Led by Woody, Andy's toys live happily in his ...
Jumanji                        When siblings Judy and Peter discover an encha...
Grumpier Old Men               A family wedding reignites the ancient feud be...
Waiting to Exhale              Cheated on, mistreated and stepped on, the wom...
Father of the Bride Part II    Just when George Banks has recovered from his ...
Name: overview, dtype: object
```

In [27]:

```
# Create tf-idf matrix for text comparison
tfidf = TfidfVectorizer(stop_words='english')
tfidf_matrix = tfidf.fit_transform(movie_metadata['overview'])
```

In [28]:

```
# Compute cosine similarity between all movie-descriptions
similarity = cosine_similarity(tfidf_matrix)
similarity_df = pd.DataFrame(similarity,
                             index=movie_metadata.index.values,
                             columns=movie_metadata.index.values)
similarity_df.head(10)
```

Out[28]:

	Toy Story	Jumanji	Grumpier Old Men	Waiting to Exhale	Father of the Bride Part II	Heat	Sabrina	Tom and Huck	Sudden Death	G
Toy Story	1.000000	0.015385	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	
Jumanji	0.015385	1.000000	0.046854	0.000000	0.000000	0.047646	0.000000	0.000000	0.098488	
Grumpier Old Men	0.000000	0.046854	1.000000	0.000000	0.023903	0.000000	0.000000	0.006463	0.000000	
Waiting to Exhale	0.000000	0.000000	0.000000	1.000000	0.000000	0.007417	0.000000	0.008592	0.000000	
Father of the Bride Part II	0.000000	0.000000	0.023903	0.000000	1.000000	0.000000	0.030866	0.000000	0.033213	
Heat	0.000000	0.047646	0.000000	0.007417	0.000000	1.000000	0.000000	0.000000	0.046349	
Sabrina	0.000000	0.000000	0.000000	0.000000	0.030866	0.000000	1.000000	0.000000	0.000000	
Tom and Huck	0.000000	0.000000	0.006463	0.008592	0.000000	0.000000	0.000000	1.000000	0.000000	
Sudden Death	0.000000	0.098488	0.000000	0.000000	0.033213	0.046349	0.000000	0.000000	1.000000	
GoldenEye	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	

10 rows × 21604 columns

In [29]:

```
# movie list
movie_list = similarity_df.columns.values


# sample movie
movie = 'Batman Begins'

# top recommendation movie count
top_n = 10

# get movie similarity records
movie_sim = similarity_df[similarity_df.index == movie].values[0]

# get movies sorted by similarity
sorted_movie_ids = np.argsort(movie_sim)[::-1]

# get recommended movie names
recommended_movies = movie_list[sorted_movie_ids[1:top_n+1]]

print('\n\nTop Recommended Movies for:', movie, 'are:-\n', recommended_movies)
```

```
Top Recommended Movies for: Batman Begins are:-
 ['Batman Unmasked: The Psychology of the Dark Knight'
 'Batman: The Dark Knight Returns, Part 1' 'Batman: Bad Blood'
 'Batman: Year One' 'Batman: Under the Red Hood'
```

```
 'Batman Beyond: The Movie' 'Batman Forever'
 'Batman: Mask of the Phantasm' 'Batman & Bill' 'Batman']
```

Your turn: Create a function as defined below, **`content_movie_recommender()`** which can take in sample movie names and print a list of top N recommended movies

In [30]:

```
def content_movie_recommender(input_movie, similarity_database=similarity_df, movie_dat

    # get movie similarity records
    movie_sim = similarity_df[similarity_df.index == input_movie].values[0]

    # get movies sorted by similarity
    sorted_movie_ids = np.argsort(movie_sim)[::-1]

    # get recommended movie names
    return movie_list[sorted_movie_ids[1:top_n+1]]
```

Your turn: Test your function below on the given sample movies

In [31]:

```
sample_movies = ['Captain America', 'The Terminator', 'The Exorcist',
                 'The Hunger Games: Mockingjay - Part 1', 'The Blair Witch Project']

for movie in sample_movies :
    sim_movies = content_movie_recommender(movie)
    print("Movies similar to ", movie)
    for sim_movie in sim_movies :
        print("    ", sim_movie)
```

```
Movies similar to  Captain America
     Iron Man & Captain America: Heroes United
     Captain America: The First Avenger
     Team Thor
     Education for Death
     Captain America: The Winter Soldier
     49th Parallel
     Ultimate Avengers
     Philadelphia Experiment II
     Vice Versa
     The Lair of the White Worm
Movies similar to  The Terminator
     Terminator 2: Judgment Day
     Terminator Salvation
     Terminator 3: Rise of the Machines
     Silent House
     They Wait
     Another World
     Teenage Caveman
     Appleseed Alpha
     Respire
     Just Married
Movies similar to  The Exorcist
     Exorcist II: The Heretic
     Domestic Disturbance
     Damien: Omen II
     The Exorcist III
     Like Sunday, Like Rain
     People Like Us
     Quand on a 17 Ans
     Don't Knock Twice
```

```
     Zero Day
     Brick Mansions
Movies similar to  The Hunger Games: Mockingjay - Part 1
     The Hunger Games: Catching Fire
     The Hunger Games: Mockingjay - Part 2
     Last Train from Gun Hill
     The Hunger Games
     Will Success Spoil Rock Hunter?
     Circumstance
     Man of Steel
     The Amityville Horror
     Pregnancy Pact
     Bananas
Movies similar to  The Blair Witch Project
     Book of Shadows: Blair Witch 2
     Freakonomics
     Le Bal des actrices
     Greystone Park
     Willow Creek
     Addio zio Tom
     The Conspiracy
     A Haunted House
     Tonight She Comes
     Curse of the Blair Witch
```

8.3: Collaborative filtering Recommendation Systems

Collaborative Filtering

Primarily recommends content to you based on inputs or actions from other people(say your friends). collaborative filtering

What is the intuition behind this?

- **Personal tastes are correlated**

```
    1.   If Alice and Bob both like X and Alice likes Y then Bob is more likely to like Y
    2.   especially (perhaps) if Bob knows Alice
```

Types of Collaborative Filtering:

1. Neighborhood methods
2. Matrix Factorization (Latent Factor) methods

Assume you dont have users. Rather you have users' characterisics and properties(as shown in image).

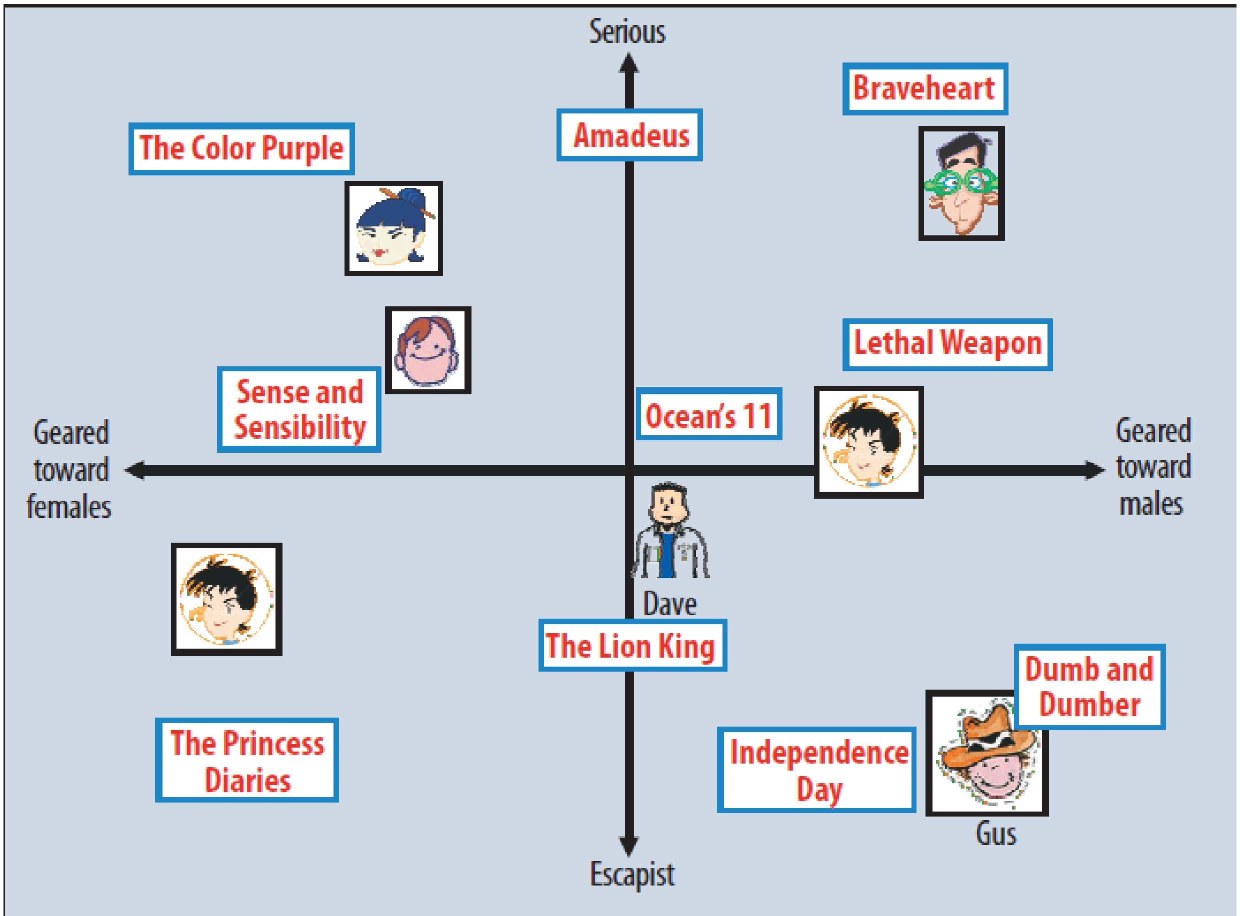


Figure 2. **A simplified illustration of the latent factor approach, which characterizes both users and movies using two axes—male versus female and serious versus escapist.**

For example, a person who is brave-hearted is more likely to be interested in dark, horrific movies rather than someone who is soft and compassionate.

- ^This is just an example(not in any literal sense)

So, once you have the properties and characteristics of each user, we call them as lower-dimensional features of the users. Similarly, we can have lower-dimensional features for movies(say its 10% action, 20% romance ...)

With these features, we represent users and movies in a low dimensional space describing their properties. **This is called as the latent space.**

We then recommend a movie based on its proximity to the user in the latent space.

The problem:

The problem we try to address here is the rating prediction problem. Say, we try to guess how much Alice would rate a movie and suggest those movies that we think Alice will rate higher.

Interesting...But, how do we predict how much Alice would rate a movie?

The data we have is a rating history: ratings of users for items in the interval [1,5]. We can put all this data into a sparse matrix called R:

$$R = \begin{array}{c} \begin{pmatrix} 3 & ? & ? \\ ? & 4 & 5 \\ ? & ? & 2 \\ 2 & 3 & ? \end{pmatrix} \end{array} \begin{array}{c} \text{Alice} \\ \text{Bob} \\ \text{Chand} \\ \text{Deb} \end{array}$$

Each row of the matrix corresponds to a given user, and each column corresponds to a given item. For instance here, Alice has rated the first movie with a rating of 3, and Chand has rated the third item with a rating of 2.

The matrix R is sparse (more than 99% of the entries are missing), and our goal is to predict the missing entries, i.e. predict the ?.

Anatomy of the Rating matrix: LATENT SPACE

Before predicting ratings, lets step back and understand the latent space more! \ In this Rating matrix, Rows represent Users and Columns represent Movies. $R = \begin{pmatrix} --\text{Alice}-- \\ --\text{Bob}-- \\ --\text{Chand}-- \\ --\text{Deb}-- \end{pmatrix}$

In latent space(low dimensional features - fanatics), for instance, Alice could be defined as a little bit of an action fan, a little bit of a comedy fan, a lot of a romance fan, etc. As for Bob, he could be more keen on action movies:

```
Alice = 10% Action fan + 10% Comedy fan + 50% Romance fan + ⋯ \\
Bob = 50% Action fan + 30% Comedy fan + 10% Romance fan + ⋯ \\
: \\
Zoe = ⋯
```

What would happen if we transposed our rating matrix? Instead of having users in the rows, we would now have movies, defined as their ratings.

$$R^T = \begin{pmatrix} --\text{Avengers}-- \\ --\text{Matrix}-- \\ --\text{Inception}-- \\ --\text{Sherlock}-- \end{pmatrix}$$

In the latent space, we will associate a semantic meaning behind each of the movies, and these semantic meanings(say movie characteristics) can build back all of our original movies.

EXAMPLE

In the below example, we convert users and movies to vectors(embeddings) and do dot-product to predict R

user vector - U \ movies vector - V \ $R = U . V$

Additional hints:

use dataframe map - map

Create tensor - Input

Create Embedding - Embedding

Dot product - Dot

Fit model : fit

Measure Performance: mean_squared_error

Q8.3: Building a Deep Learning Matrix Factorization based Collaborative Filtering Recommendation System

Your Turn: Fill in the necessary blank code snippets in the following sections to train your own DL collaborative filtering system

Create Configuration Parameters

In [32]:

```
# Create user and movie-id mapping to convert to numbers
user_id_mapping = {id:i for i, id in enumerate(df_filtered['User'].unique())}
movie_id_mapping = {id:i for i, id in enumerate(df_filtered['Movie'].unique())}
```

In [33]:

```
# use dataframe map function to map users & movies to mapped ids based on above mapping
train_user_data = df_train['User'].map(user_id_mapping)
train_movie_data = df_train['Movie'].map(movie_id_mapping)
```

In [34]:

```
# do the same for test data
test_user_data = df_test['User'].map(user_id_mapping)
test_movie_data = df_test['Movie'].map(movie_id_mapping)
```

In [35]:

```
# Get input variable-sizes
users = len(user_id_mapping)
movies = len(movie_id_mapping)
embedding_size = 100
```

Construct Deep Learning Model Architecture

In [36]:

```
# use Input() to create tensors for - 'user' and 'movie'
user_id_input = Input(shape=(1,), name='user')
movie_id_input = Input(shape=(1,), name='movie')
```

In [37]:

```
# Create embedding layer for users
user_embedding = Embedding(output_dim=embedding_size,
                           input_dim=users,
                           input_length=1,
```

```
                           name='user_embedding')(user_id_input)

# create embedding layer for movies just like users
movie_embedding = Embedding(output_dim=embedding_size,
                           input_dim=movies,
                           input_length=1,
                           name='movie_embedding')(movie_id_input)
```

In [38]:

```
# Reshape the embedding layers
user_vector = Reshape([embedding_size])(user_embedding)
movie_vector = Reshape([embedding_size])(movie_embedding)
```

In [39]:

```
# Compute dot-product of reshaped embedding layers as prediction
y = Dot(1, normalize=False)([user_vector, movie_vector])
```

In [40]:

```
# Setup model
model = Model(inputs=[user_id_input, movie_id_input], outputs=y)
model.compile(loss='mse', optimizer='adam')
model.summary()
```

```
Model: "model_1"
__________________________________________________________________________________________
________
Layer (type)                    Output Shape         Param #     Connected to
==========================================================================================
==========
user (InputLayer)               (None, 1)            0
__________________________________________________________________________________________
________
movie (InputLayer)              (None, 1)            0
__________________________________________________________________________________________
________
user_embedding (Embedding)      (None, 1, 100)       2082800     user[0][0]
__________________________________________________________________________________________
________
movie_embedding (Embedding)     (None, 1, 100)       174100      movie[0][0]
__________________________________________________________________________________________
________
reshape_1 (Reshape)             (None, 100)          0           user_embedding[0][0]
__________________________________________________________________________________________
________
reshape_2 (Reshape)             (None, 100)          0           movie_embedding[0][0]
__________________________________________________________________________________________
________
dot_1 (Dot)                     (None, 1)            0           reshape_1[0][0]
                                                                 reshape_2[0][0]
==========================================================================================
==========
Total params: 2,256,900
Trainable params: 2,256,900
Non-trainable params: 0
__________________________________________________________________________________________
________
```

Train and Test the Model

In [41]:
```
# Fit model
X = [train_user_data, train_movie_data]
y = df_train['Rating']

batch_size = 1024
epochs = 5
validation_split = 0.1

model.fit(X, y,
          batch_size=batch_size,
          epochs=epochs,
          validation_split=validation_split,
          shuffle=True,
          verbose=1)
```

```
Train on 5247522 samples, validate on 583059 samples
Epoch 1/5
5247522/5247522 [==============================] - 373s 71us/step - loss: 2.0790 - val_l
oss: 0.7862
Epoch 2/5
5247522/5247522 [==============================] - 346s 66us/step - loss: 0.7458 - val_l
oss: 0.7307oss
Epoch 3/5
5247522/5247522 [==============================] - 252s 48us/step - loss: 0.6780 - val_l
oss: 0.7006
Epoch 4/5
5247522/5247522 [==============================] - 161s 31us/step - loss: 0.6140 - val_l
oss: 0.6896
Epoch 5/5
5247522/5247522 [==============================] - 164s 31us/step - loss: 0.5501 - val_l
oss: 0.7002
```

Out[41]:
```
<keras.callbacks.History at 0x173a7375fc8>
```

In [42]:
```
# Test model by making predictions on test data
y_pred = model.predict([test_user_data, test_movie_data]).ravel()
# clip upper and lower ratings
y_pred = list(map(lambda x: 1.0 if x < 1 else 5.0 if x > 5.0 else x, y_pred))
# get true labels
y_true = df_test['Rating'].values

#  Compute RMSE
rmse = np.sqrt(mean_squared_error(y_pred=y_pred, y_true=y_true))
print('\n\nTesting Result With DL Matrix-Factorization: {:.4f} RMSE'.format(rmse))
```

```
Testing Result With DL Matrix-Factorization: 0.8335 RMSE
```

In [43]:
```
## Let's see how our collaborative model performs by seeing the predicted and actual ra
results_df = pd.DataFrame({
    'User ID': test_user_data.values,
    'Movie ID': test_movie_data.values,
    'Movie Name': [movie_titles['Name'].iloc[item] for item in test_movie_data],
    'Predicted Rating': np.round(y_pred, 1),
    'Actual Rating': y_true
})

results_df.head(20)
```

Out[43]:

	User ID	Movie ID	Movie Name	Predicted Rating	Actual Rating
0	20817	928	Journeys with George	3.1	3.0
1	18236	521	Love Songs	4.3	5.0
2	10516	126	Fatal Beauty	3.6	3.0
3	6528	231	Gross Anatomy	4.9	4.0
4	17831	1369	Marathon Man	2.9	3.0
5	19722	1034	Disclosure	3.0	3.0
6	13217	1089	Eel	4.3	3.0
7	2025	395	Arjuna: Complete Collection	1.4	1.0
8	16947	824	Bill Cosby: Himself	3.3	4.0
9	2	622	Dario Argento Collection: Vol. 2: Demons 2	3.3	2.0
10	18262	537	A Crime of Passion	4.4	3.0
11	7114	423	Happiness	3.0	1.0
12	14022	425	Recess: School's Out	2.7	3.0
13	1670	204	Troy: Bonus Material	3.9	5.0
14	864	440	Dark Shadows: Vol. 9	3.6	4.0
15	13117	82	Silkwood	4.0	4.0
16	10467	220	Voyage to the Planets and Beyond	3.5	4.0
17	1560	100	Complete Shamanic Princess	4.4	5.0
18	1217	1008	Judaai	2.9	2.0
19	1698	817	Logan's Run	3.7	2.0

8.4: Hybrid Recommendation System (Content & Collaborative)

One advantage of deep learning models is, that movie-metadata can easily be added to the model. We will tf-idf transform the short description of all movies to a sparse vector. The model will learn to reduce the dimensionality of this vector and how to combine metadata with the embedding of the user-id and the movie-id. In this way we can add any additional metadata to our own recommender. These kind of hybrid systems can learn how to reduce the impact of the cold start problem.

Deep learning models require lots of data to train and predict. To provide our model with more data, we will include the movie metadata as well. We will do the following:

- Use movie metadata to combine with user and movie matrices in order to get more data
- Use tf-idf transform to vectorize movie metadata (Sparse Layer)
- Create an embedding of the metadata 512 -> 256
- Combine all embeddings for movie tf-idf vectors, user and ratings to arrive at a common embedding space (256 sized embeddings per entity)

- Use the embeddings to train the model and get predictions on the test data

Additional Hints:

Dense layer setup : Dense

Create model using tf.keras API : Model

Compile model using : Compile

Fit model : fit

Predict accuracy: mean_squared_error

Q8.3: Building a Deep Learning Hybrid Recommendation System

We will be building the following hybrid deep learning recommendation model as scene in the following schematic.

Your Turn: Fill in the necessary blank code snippets in the following sections to train your own DL hybrid recommendation system

Create Configuration Parameters

In [44]:

```
# ceate a copy of the filtered data frame
df_filtered_cp = df_filtered.copy(deep=True)
```

In [45]:

```
# Create user- & movie-id mapping
user_id_mapping = {id:i for i, id in enumerate(df_filtered_cp['User'].unique())}
movie_id_mapping = {id:i for i, id in enumerate(df_filtered_cp['Movie'].unique())}
```

In [46]:

```
# use dataframe map function to map users & movies to mapped ids based on above mapping
df_filtered_cp['User'] = df_filtered_cp['User'].map(user_id_mapping)
df_filtered_cp['Movie'] = df_filtered_cp['Movie'].map(movie_id_mapping)
```

Create Movie Description Dataset (Content)

In [47]:

```
# Preprocess metadata
tmp_metadata = movie_metadata.copy()
tmp_metadata.index = tmp_metadata.index.str.lower()

# Preprocess titles
tmp_titles = movie_titles.drop('Year', axis=1).copy()
tmp_titles = tmp_titles.reset_index().set_index('Name')
tmp_titles.index = tmp_titles.index.str.lower()

# Combine titles and metadata
df_id_descriptions = tmp_titles.join(tmp_metadata).dropna().set_index('Id')
df_id_descriptions['overview'] = df_id_descriptions['overview'].str.lower()
```

```
#del tmp_metadata,tmp_titles
print('Movie Description DF Shape:', df_id_descriptions.shape)
df_id_descriptions.tail()
```

```
Movie Description DF Shape: (6939, 1)
```

Out[47]:

Id	overview
16182	daryl zero is a private investigator. along wi...
15233	clear the runway for derek zoolander, vh1's th...
1210	a newly arrived governor finds his province un...
17631	in 1879, during the zulu wars, man of the peop...
17631	as a child, ali neuman narrowly escaped being ...

Create User-Rating Filtered Dataset (Collaborative)

Here we filter out movie-user-ratings where movies don't have descriptions (content)

In [48]:

```
df_hybrid = (df_filtered_cp.set_index('Movie')
               .join(df_id_descriptions)
               .dropna()
               .drop('overview', axis=1)
               .reset_index().rename({'index':'Movie'},
                                     axis=1))
print('Movie-User-Rating DF Shape:', df_hybrid.shape)
df_hybrid.head()
```

```
Movie-User-Rating DF Shape: (2286494, 3)
```

Out[48]:

	Movie	User	Rating
0	12	12	1.0
1	12	966	4.0
2	12	1518	2.0
3	12	9032	4.0
4	12	11849	5.0

In [49]:

```
# Split train- & testset
n = 300000
df_hybrid = df_hybrid.sample(frac=1).reset_index(drop=True)
df_hybrid_train = df_hybrid[:-n]
df_hybrid_test = df_hybrid[-n:]
df_hybrid_train.shape, df_hybrid_test.shape
```

Out[49]:

```
((1986494, 3), (300000, 3))
```

Generate TFIDF Vectors for Train and Test Datasets (Movie Descriptions)

In [50]:

```
# Create tf-idf matrix for movie description vectors - HINT: check the overview column
```

```
tfidf = TfidfVectorizer(stop_words='english')
tfidf_hybrid = tfidf.fit_transform(df_id_descriptions['overview'])
```

In [51]:
```
# Get mapping from movie-ids to indices in tfidf-matrix
movie_idx_mapping = {id:i for i, id in enumerate(df_id_descriptions.index)}
```

In [52]:
```
# get train data tfidf vectors
train_tfidf = []

# Iterate over all movie-ids and save the tfidf-vectors (sparse format for memory effic
for idx in tqdm(df_hybrid_train['Movie'].values):
    index = movie_idx_mapping[idx]
    train_tfidf.append(tfidf_hybrid[index])

len(train_tfidf)
```

```
100%|██████████| 1986494/1986494 [03:11<00:00, 10363.82it/s]
```

Out[52]:
```
1986494
```

In [53]:
```
# get test data tfidf vectors
test_tfidf = []

# Iterate over all movie-ids and save the tfidf-vectors (sparse format for memory effic
for idx in tqdm(df_hybrid_test['Movie'].values):
    index = movie_idx_mapping[idx]
    test_tfidf.append(tfidf_hybrid[index])

len(test_tfidf)
```

```
100%|██████████| 300000/300000 [00:57<00:00, 5250.32it/s]
```

Out[53]:
```
300000
```

In [54]:
```
# Stack the sparse matrices
train_tfidf = vstack(train_tfidf)
test_tfidf = vstack(test_tfidf)

train_tfidf.shape, test_tfidf.shape
```

Out[54]:
```
((1986494, 24144), (300000, 24144))
```

In [55]:
```
type(train_tfidf)
```

Out[55]:
```
scipy.sparse.csr.csr_matrix
```

This shows we are using sparse matrices to represent the vectors as dense vectors would typically give a out of memory error!

Construct Deep Learning Model Architecture

In [56]:
```
# setup NN parameters
user_embed_dim = 256
```

```
movie_embed_dim = 256
userid_input_shape = 1
movieid_input_shape = 1
tfidf_input_shape = tfidf_hybrid.shape[1]
```

In [57]:

```
# Create the input layers

# user and movie input layers
user_id_input = Input(shape=(userid_input_shape,), name='user')
movie_id_input = Input(shape=(movieid_input_shape,), name='movie')

# tfidf input layer
tfidf_input = Input(shape=(tfidf_input_shape,), name='tfidf', sparse=True)
```

In [58]:

```
# Create embeddings layers for users and movies

# user embedding
user_embedding = Embedding(output_dim=user_embed_dim,
                           input_dim=len(user_id_mapping),
                           input_length=userid_input_shape,
                           name='user_embedding')(user_id_input)

# movie embedding
movie_embedding = Embedding(output_dim=movie_embed_dim,
                            input_dim=len(movie_id_mapping),
                            input_length=movieid_input_shape,
                            name='movie_embedding')(movie_id_input)
```

In [59]:

```
# Dimensionality reduction with Dense layers
tfidf_vectors = Dense(512, activation='relu')(tfidf_input)
tfidf_vectors = Dense(256, activation='relu')(tfidf_vectors)
```

In [60]:

```
# Reshape both user and movie embedding layers
user_vectors = Reshape([user_embed_dim])(user_embedding)
movie_vectors = Reshape([movie_embed_dim])(movie_embedding)
```

In [61]:

```
# Concatenate all layers into one
hybrid_layer = Concatenate()([user_vectors, movie_vectors, tfidf_vectors])
```

In [62]:

```
# add in dense and output layers
dense = Dense(512, activation='relu')(hybrid_layer)
dense = Dropout(0.2)(dense)
output = Dense(1)(dense)
```

In [63]:

```
# create and view model summary
model = Model(inputs=[user_id_input, movie_id_input, tfidf_input], outputs=output)
model.compile(loss='mse', optimizer='adam')
model.summary()
```

```
Model: "model_2"
```

```
__________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
user (InputLayer)               (None, 1)            0
__________________________________________________________________________________________________
movie (InputLayer)              (None, 1)            0
__________________________________________________________________________________________________
tfidf (InputLayer)              (None, 24144)        0
__________________________________________________________________________________________________
user_embedding (Embedding)      (None, 1, 256)       5331968     user[0][0]
__________________________________________________________________________________________________
movie_embedding (Embedding)     (None, 1, 256)       445696      movie[0][0]
__________________________________________________________________________________________________
dense_1 (Dense)                 (None, 512)          12362240    tfidf[0][0]
__________________________________________________________________________________________________
reshape_3 (Reshape)             (None, 256)          0           user_embedding[0][0]
__________________________________________________________________________________________________
reshape_4 (Reshape)             (None, 256)          0           movie_embedding[0][0]
__________________________________________________________________________________________________
dense_2 (Dense)                 (None, 256)          131328      dense_1[0][0]
__________________________________________________________________________________________________
concatenate_1 (Concatenate)     (None, 768)          0           reshape_3[0][0]
                                                                 reshape_4[0][0]
                                                                 dense_2[0][0]
__________________________________________________________________________________________________
dense_3 (Dense)                 (None, 512)          393728      concatenate_1[0][0]
__________________________________________________________________________________________________
dropout_1 (Dropout)             (None, 512)          0           dense_3[0][0]
__________________________________________________________________________________________________
dense_4 (Dense)                 (None, 1)            513         dropout_1[0][0]
==================================================================================================
Total params: 18,665,473
Trainable params: 18,665,473
Non-trainable params: 0
__________________________________________________________________________________________________
```

Train and Test the Model

In [64]:

```
# fit the model
batch_size=1024
epochs=10
X = [df_hybrid_train['User'], df_hybrid_train['Movie'], train_tfidf]
y = df_hybrid_train['Rating']
```

```
print(type(X[0]))
print(type(y))

model.fit(X, y,
          batch_size=batch_size,
          epochs=epochs, ## Change the epochs to find better improved model.
          validation_split=0.1,
          shuffle=True)
```

```
<class 'pandas.core.series.Series'>
<class 'pandas.core.series.Series'>
Train on 1787844 samples, validate on 198650 samples
Epoch 1/10
1787844/1787844 [==============================] - 991s 554us/step - loss: 0.9323 - val_
loss: 0.8146
Epoch 2/10
1787844/1787844 [==============================] - 951s 532us/step - loss: 0.7987 - val_
loss: 0.7675
Epoch 3/10
1787844/1787844 [==============================] - 576s 322us/step - loss: 0.7642 - val_
loss: 0.7454
Epoch 4/10
1787844/1787844 [==============================] - 575s 321us/step - loss: 0.7297 - val_
loss: 0.7216
Epoch 5/10
1787844/1787844 [==============================] - 613s 343us/step - loss: 0.6951 - val_
loss: 0.7011
Epoch 6/10
1787844/1787844 [==============================] - 681s 381us/step - loss: 0.6599 - val_
loss: 0.6843
Epoch 7/10
1787844/1787844 [==============================] - 733s 410us/step - loss: 0.6255 - val_
loss: 0.6707
Epoch 8/10
1787844/1787844 [==============================] - 643s 359us/step - loss: 0.5940 - val_
loss: 0.6632
Epoch 9/10
1787844/1787844 [==============================] - 550s 308us/step - loss: 0.5643 - val_
loss: 0.6587
Epoch 10/10
1787844/1787844 [==============================] - 618s 346us/step - loss: 0.5385 - val_
loss: 0.6558
```

Out[64]:
```
<keras.callbacks.History at 0x1738029f5c8>
```

In [65]:
```
# create test input data and true outputs
X_test = [df_hybrid_test['User'], df_hybrid_test['Movie'], test_tfidf]
y_true = df_hybrid_test['Rating'].values

# Test model by making predictions on test data
y_pred = model.predict(X_test).ravel()
# clip upper and lower ratings
y_pred = list(map(lambda x: 1.0 if x < 1 else 5.0 if x > 5.0 else x, y_pred))

#  Compute RMSE
rmse = np.sqrt(mean_squared_error(y_pred=y_pred, y_true=y_true))
print('\n\nTesting Result With DL Hybrid Recommender: {:.4f} RMSE'.format(rmse))
```

```
Testing Result With DL Hybrid Recommender: 0.8115 RMSE
```

In [66]:

```python
## Let's see how our collaborative model performs by seeing the predicted and actual ra
results_df = pd.DataFrame({
    'User ID': df_hybrid_test['User'].values,
    'Movie ID': df_hybrid_test['Movie'].values,
    'Movie Name': [movie_titles['Name'].iloc[item] for item in df_hybrid_test['Movie']]
    'Predicted Rating': np.round(y_pred, 1),
    'Actual Rating': y_true
})

results_df.head(20)
```

Out[66]:

	User ID	Movie ID	Movie Name	Predicted Rating	Actual Rating
0	13302	985	The Trip	2.8	3.0
1	6233	197	Gupt	3.3	1.0
2	9810	384	The Santa Clause 2	1.4	1.0
3	11936	798	Teen Titans: Season 1	3.0	3.0
4	3225	1027	The Educational Archives: Vol. 1: Sex & Drugs	3.0	3.0
5	17533	313	Saturday Night Live: The Best of Jon Lovitz	2.8	3.0
6	20812	1511	Blue's Clues: Blue's Room: Beyond Your Wildest...	3.2	3.0
7	17464	212	Dinner Rush	2.4	5.0
8	5293	1062	Hemp Revolution	1.9	3.0
9	12740	524	The Last Seduction II	2.3	2.0
10	280	831	Tupac: Resurrection	4.2	1.0
11	14536	235	Cartoon Crazys Sci-Fi	3.5	5.0
12	1449	501	Mitch Hedberg: Mitch All Together	2.9	2.0
13	2387	1392	Mr. Murder	3.3	3.0
14	14582	940	Screw Loose	3.8	4.0
15	20234	1250	Frankenthumb	2.4	4.0
16	20174	1140	Kyun! Ho Gaya Na	2.6	5.0
17	16224	269	Sex and the City: Season 4	3.1	1.0
18	729	384	The Santa Clause 2	3.7	4.0
19	13643	843	La Cienaga	3.3	3.0