MASTER 2 MANAGEMENT AND INTERNATIONAL BUSINESS

BIG DATA COURSE

"Movie Recommendation System"

Students:

CHALUNGTSANG Tenzin
KHIN Vannkiriya
LIU Kelly
PHORIMAVONG Gwendoline
TRINH Laurie

Professor: KHALFALLAH Malik
Academic year: 2024/2025

Introduction

Good morning, today we'll present our project on movie recommendation systems. We used a Kaggle notebook titled *'Getting Started with a Movie Recommendation System'*, which implements a content-based approach.

The main goal is to suggest similar movies to users by analyzing movie metadata, such as genre, director, actors, and description. To achieve this, we use a dataset containing movies released up to July 2017 with detailed information such as cast, production team, plot keywords, budget, revenue, posters, and even release dates.

Through this study, we aim to understand how recommendation systems use this information to generate relevant suggestions and apply commonly used techniques in this field.

I. We going to start with the first part : The Original Notebook

The datasets used in the Kaggle notebook are files containing information about movies, users, and movie ratings. They help train the recommendation model by providing a database from which it can learn to make predictions.

A. TMDB 5000 five thousand Movies

The first dataset, tmdb_5000_movies.dot csv, contains details about the movies themselves, such as title, genre, duration, and popularity.
For example, *Inception* is a science fiction movie that lasts 148 and has a high popularity score.
This information helps understand viewer preferences based on the types of movies they watch.

B. TMDB 5000 Credits

The second dataset, tmdb_5000_credits.csv, contains information about actors and directors.
For example, it shows that Leonardo DiCaprio played in *Titanic* and that the director of the movie is James Cameron.
This file is important for analyzing connections between the people involved in a movie and its success.

C. MovieLens Ratings (Small)

The third dataset, ratings_small.csv, contains ratings given by users to different movies.
For example, if a user watched a movie and rated it 4 out of 5 stars, this information is stored in this file.
These ratings are important for the model to learn which movies are liked and which are not, so it can make better recommendations. By using these three datasets, the model can be trained with movie information, cast and crew details, and audience ratings, helping it predict which movies a user might enjoy.

The first two datasets are used for content-based recommendation, while the last one allows us to implement collaborative filtering using the SVD model. These three dataset from the backbone of our movie recommendation system allowing us to analyze and predict movie preferences effectively.

II. Problems we had

Now, I'm gonna talk about some of the problems we had :

When we tried to run the original notebook code on Google Colab, we had many problems. On Kaggle, everything is ready to use, but on Colab, the datasets are not recognized automatically. So, we had to download the files from Kaggle and make them available in Colab before using them.

A. Accessing to files

In this project, we needed to access movie data from Kaggle. On Kaggle, the files are already available in a specific folder, so we can open them easily using a simple command like pd.read_csv(). However, on Google Colab, these files are not there by default, so we had to find another way to get them. To solve this, we used the Kaggle API, a tool that lets us download files automatically. First, we installed the Kaggle library and added a special file (kaggle.json) that contains a key to connect to Kaggle. After adjusting the settings to allow access, we downloaded the files. Since they were compressed in a ZIP folder, we had to unzip them before using them.

Our code works on both Kaggle and Colab because we used the os module, which helps us find and open the files in the right location. This way, we can access the data without any issues, no matter where we run the code.

B. Access to the Surprise library

Another issue we ran into : The second problem was that scikit-surprise was not installed by default on Colab, while it was on Kaggle. To fix this, we had to install it manually by running a command that downloaded and installed the tool for us.

In addition, we also had to deal with a difference in where the files were stored. On Kaggle, the files were placed in a specific folder, but on Colab, they were just in the main folder. Because of this, we had to adjust our code to look for the files in the right place depending on the platform we were using.

To give a simple way to think about this, it’s kind of like looking for a book in a library.is like trying to find a book in a library. On Kaggle, the book is always kept in the same section, so it’s easy to find. But on Colab, the book could be anywhere, so we had to check the location and tell our program exactly where to look.

C. The split(n_folds=5) function does not exist anymore

Finally, changes in the Surprise library functions

The last problem we encountered was with the Surprise library, which we used to make movie recommendations. First, we had to manually download the dataset because it wasn’t automatically available. Then, we noticed that some parts of the code didn’t work because certain functions had changed.

One example is the function evaluate(), which was used in the original code but no longer exists. We had to replace it with cross_validate(), which is now the correct way to check if our model is working well. Another issue was with the split(n_folds=5) function, which was originally used to divide the data for testing. This function was removed, so we had to use cross_validate() instead, as it performs the same task.

It's like following an old recipe where some ingredients are no longer available. We had to find the new versions of those ingredients to make sure the recipe still worked.

Summary

We had to make all these changes to make the notebook work on Google Colab. We needed to set up file access, update some functions to match the latest library versions, and install all necessary dependencies.

III. Models

This notebook contains two machine learning models, specifically recommendation systems. These models are designed to suggest movies based on either their content or the preferences of other users. The goal is to create a personalized and accurate recommendation for each user.

A. Content-Based Filtering Model

The content-base filtering. this model works by recommending movies similar to those a user has already watched, based only on their descriptions. In our model, we analyze elements such as the title, synopsis, and genre of movies.

1. TF-IDF Vectorizer (TfidfVectorizer from sklearn.feature_extraction.text)

To compare movies, we use a method called TF-IDF, which stands for Term Frequency - Inverse Document Frequency. This method helps us turn movie descriptions into numbers so we can compare them more easily. The idea is simple: words that appear in almost every movie description, like "movie" or "story," are not very useful for making comparisons, so their importance is reduced. On the other hand, words that are more specific to a movie, like "action," "comedy," or "drama," are given more importance. This way, the method helps us focus on what makes each movie different. We chose the TF-IDF model because it is less resource-intensive compared to current methods, which require high computational power that is often paid on platforms like Google Colab. TF-IDF allows for efficient movie comparisons while being more resource-friendly.

2. Cosine Similarity (linear_kernel from sklearn.metrics.pairwise)

To check if two movies are similar, we use a method called "cosine similarity." This helps us compare movies by looking at the angle between their descriptions. If the angle is small, it means the movies are very similar. Instead of using a complicated calculation, we use a function called "linear_kernel," which gives the same result but is much faster. When we combine this method with TF-IDF, we create a table that shows how similar each movie is to the others. This allows us to recommend movies based on their content, without needing ratings from other users.

B. Collaborative Filtering Model

Collaborative filtering relies on comparing user ratings. Imagine a user-movie matrix where each row represents a user, and each column represents a movie. Each cell in this matrix contains the rating a user has given to a movie. For example, if User A rated Movie X with a 4 and User B rated the same movie with a 5, we can infer that they have similar tastes for that movie.

But in reality, most users have only rated a few movies, so this table has a lot of empty spaces. This makes it hard to compare users directly. To solve this, we use a method called Singular Value Decomposition (SVD), which helps us fill in the gaps and make better recommendations.

1. **SVD (Singular Value Decomposition, SVD from Surprise)**

The idea behind SVD is to decompose this user-movie matrix into several smaller matrices. Instead of working with a giant matrix full of missing values, we create smaller matrices that capture hidden relationships between users and movies. These hidden relationships often correspond to hidden factors. For example, one factor might represent "interest in action movies" or "love for romantic comedies."

These smaller matrices help us understand what types of movies a user likes, even if they haven't rated those movies yet. Thanks to SVD, the system can predict the rating a user would give to a movie they have never seen, based on the preferences of similar users. For example, if you have enjoyed several action movies that other users also liked, the system can predict that you will enjoy another action movie you haven't watched yet.

2. **Cross-Validation with cross_validate() (surprise.model_selection)**

Once we have created the SVD model, we need to check if it makes good predictions. To do this, we use a method called cross-validation. This means we divide the data into different parts, train the model on some parts, and test it on others. This helps us see if the model can correctly guess the ratings for movies a user hasn't seen yet. We also use cross-validation to ensure the model's reliability and performance, even if the data changes slightly.

In our code, we first load the user rating data from the file ratings_small.csv. Then, we create the SVD model, which analyzes the data and learns what users like. After that, we test the model using the cross_validate() function. This function splits the data into different sets and checks how well the model performs. We measure its accuracy using two main metrics. The first one, RMSE (Root Mean Squared Error), tells us how far off the predictions are from the actual ratings. The second one, MAE (Mean Absolute Error), calculates the average difference between the predicted and real ratings. Once we have tested the model, we can use it to predict how much a user would like a movie they haven't rated yet.

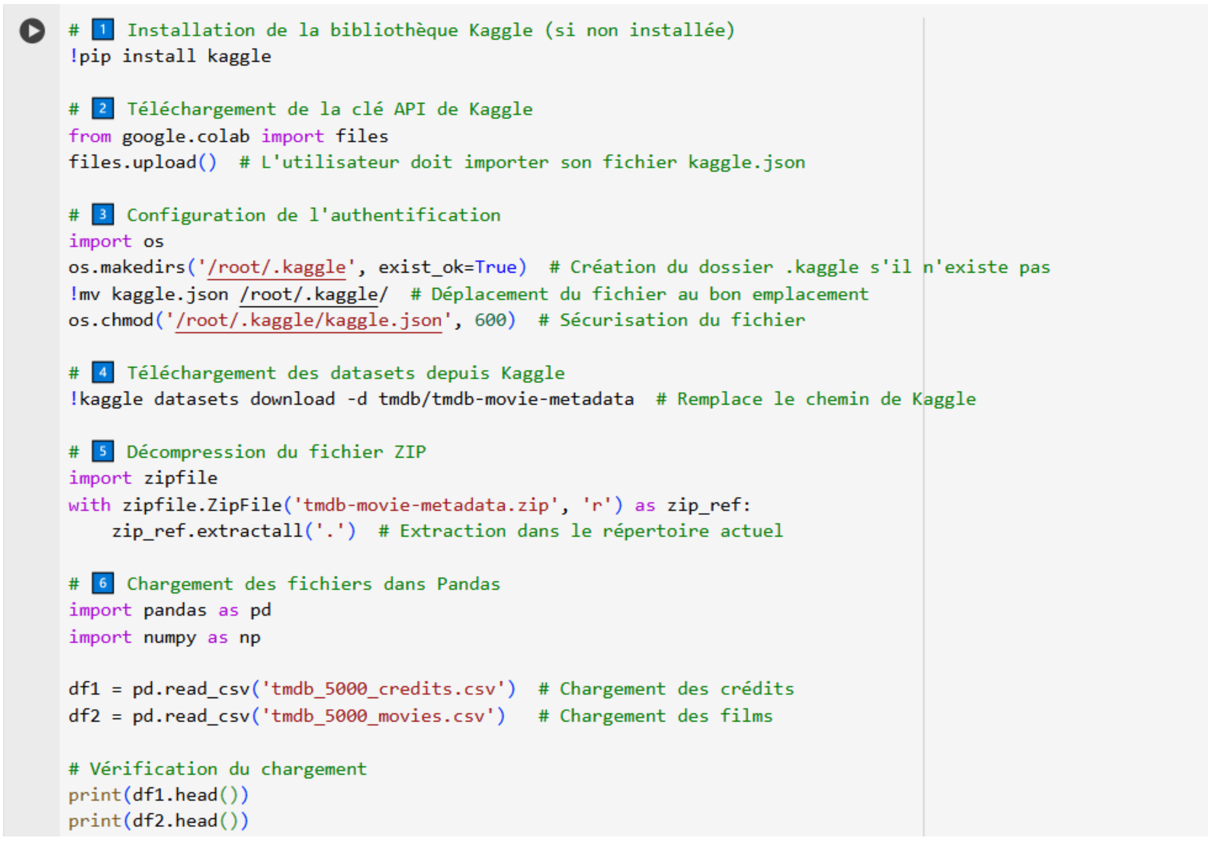
Summary

Each model has pros and cons: content-based suggests similar movies but fails for new users, while collaborative filtering (SVD) learns preferences but needs lots of data. Combining both offers better recommendations—personalized suggestions (SVD) and similar movies (TF-IDF + cosine).

IV. A look at the notebook

This notebook helps us analyze and recommend movies using three different methods: popularity, content, and collaborative filtering. Each method works in a different way and has its own advantages to help suggest movies that match what users might like.

A. Data loading and preparation



```
# 1️⃣ Installation de la bibliothèque Kaggle (si non installée)
!pip install kaggle

# 2️⃣ Téléchargement de la clé API de Kaggle
from google.colab import files
files.upload()  # L'utilisateur doit importer son fichier kaggle.json

# 3️⃣ Configuration de l'authentification
import os
os.makedirs('/root/.kaggle', exist_ok=True)  # Création du dossier .kaggle s'il n'existe pas
!mv kaggle.json /root/.kaggle/  # Déplacement du fichier au bon emplacement
os.chmod('/root/.kaggle/kaggle.json', 600)  # Sécurisation du fichier

# 4️⃣ Téléchargement des datasets depuis Kaggle
!kaggle datasets download -d tmdb/tmdb-movie-metadata  # Remplace le chemin de Kaggle

# 5️⃣ Décompression du fichier ZIP
import zipfile
with zipfile.ZipFile('tmdb-movie-metadata.zip', 'r') as zip_ref:
    zip_ref.extractall('.')  # Extraction dans le répertoire actuel

# 6️⃣ Chargement des fichiers dans Pandas
import pandas as pd
import numpy as np

df1 = pd.read_csv('tmdb_5000_credits.csv')  # Chargement des crédits
df2 = pd.read_csv('tmdb_5000_movies.csv')   # Chargement des films

# Vérification du chargement
print(df1.head())
print(df2.head())
```

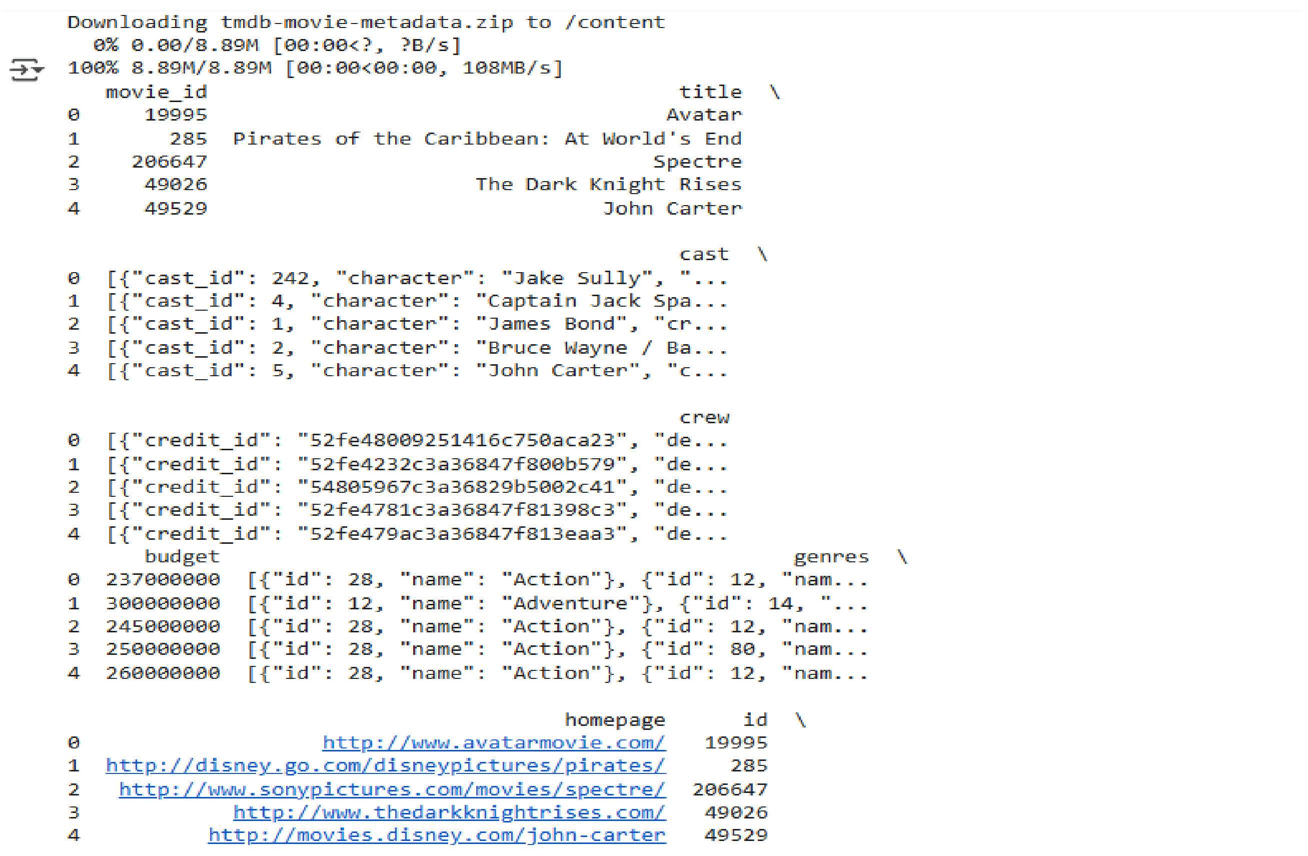
To use data from Kaggle, we first need to download it and open it in a format that we can work with using Pandas. For this, we install the **Kaggle** library, which helps us get files automatically. Then, we add a special file called **kaggle.json**, which acts as a key to connect to Kaggle. This file is stored in a hidden **.kaggle** folder, and we change its settings to keep it secure.

Once the connection is set up, we download the dataset as a **ZIP file**. Since ZIP files are compressed, we need to extract them using **zipfile.ZipFile** before we can read the data. After extracting, we open the CSV files: **tmdb_5000_credits.csv** (which contains information about actors and directors) and **tmdb_5000_movies.csv** (which has general details like movie titles, ratings, and descriptions). We store these files in two **Pandas DataFrames**:

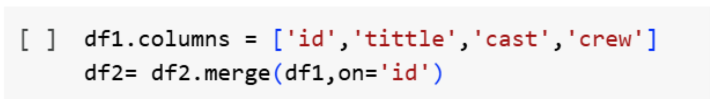
- **df1** for the credits file (actors and directors)

- **df2** for the movies file (titles, ratings, descriptions)

Finally, we check that everything is loaded correctly by displaying the first few rows of the data.



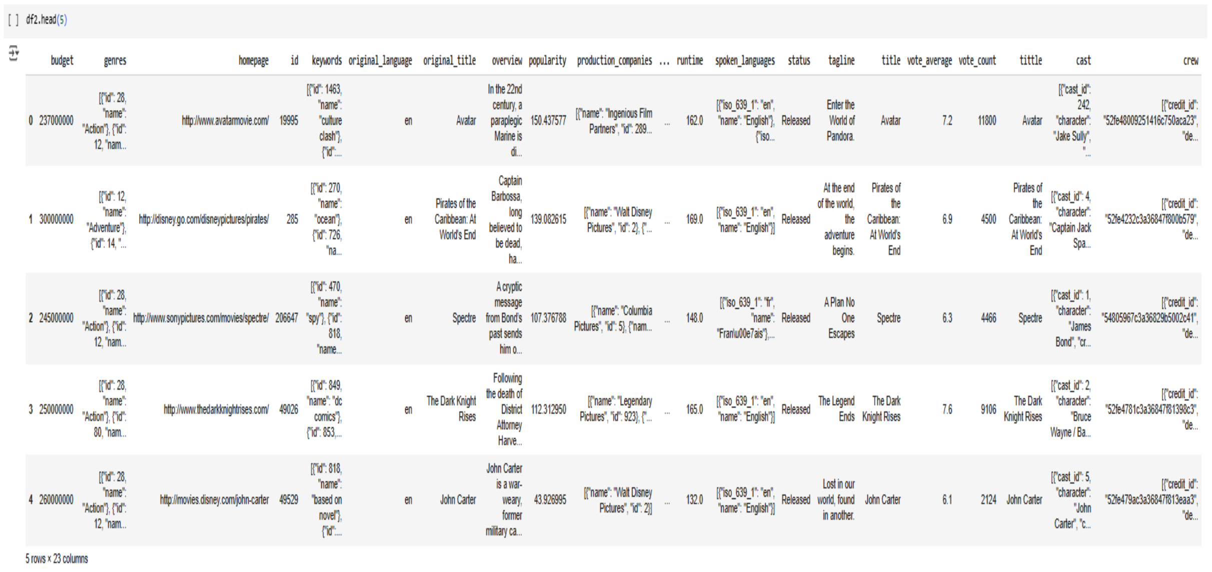
```
Downloading tmdb-movie-metadata.zip to /content
  0% 0.00/8.89M [00:00<?, ?B/s]
100% 8.89M/8.89M [00:00<00:00, 108MB/s]
   movie_id                                     title  \
0     19995                                    Avatar
1       285  Pirates of the Caribbean: At World's End
2    206647                                   Spectre
3     49026                     The Dark Knight Rises
4     49529                               John Carter

                                                cast  \
0  [{"cast_id": 242, "character": "Jake Sully", "...
1  [{"cast_id": 4, "character": "Captain Jack Spa...
2  [{"cast_id": 1, "character": "James Bond", "cr...
3  [{"cast_id": 2, "character": "Bruce Wayne / Ba...
4  [{"cast_id": 5, "character": "John Carter", "c...

                                                crew
0  [{"credit_id": "52fe48009251416c750aca23", "de...
1  [{"credit_id": "52fe4232c3a36847f800b579", "de...
2  [{"credit_id": "54805967c3a36829b5002c41", "de...
3  [{"credit_id": "52fe4781c3a36847f81398c3", "de...
4  [{"credit_id": "52fe479ac3a36847f813eaa3", "de...
      budget                                             genres  \
0  237000000  [{"id": 28, "name": "Action"}, {"id": 12, "nam...
1  300000000  [{"id": 12, "name": "Adventure"}, {"id": 14, "...
2  245000000  [{"id": 28, "name": "Action"}, {"id": 12, "nam...
3  250000000  [{"id": 28, "name": "Action"}, {"id": 80, "nam...
4  260000000  [{"id": 28, "name": "Action"}, {"id": 12, "nam...

                                       homepage      id  \
0                   http://www.avatarmovie.com/   19995
1  http://disney.go.com/disneypictures/pirates/     285
2   http://www.sonypictures.com/movies/spectre/  206647
3            http://www.thedarkknightrises.com/   49026
4          http://movies.disney.com/john-carter   49529
```



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

To analyze movies along with their actors and directors, we need to combine the two datasets. We do this by merging the two **Pandas DataFrames** using the **id** column, which is present in both files and helps match the correct movie information.

The **merge()** function links the movie details from **df2** (titles, ratings, descriptions) with the credits from **df1** (actors, directors). After merging, each row in the new DataFrame contains all the information about a movie in one place, making it easier to analyze.



```
[ ] df2.head(5)
```

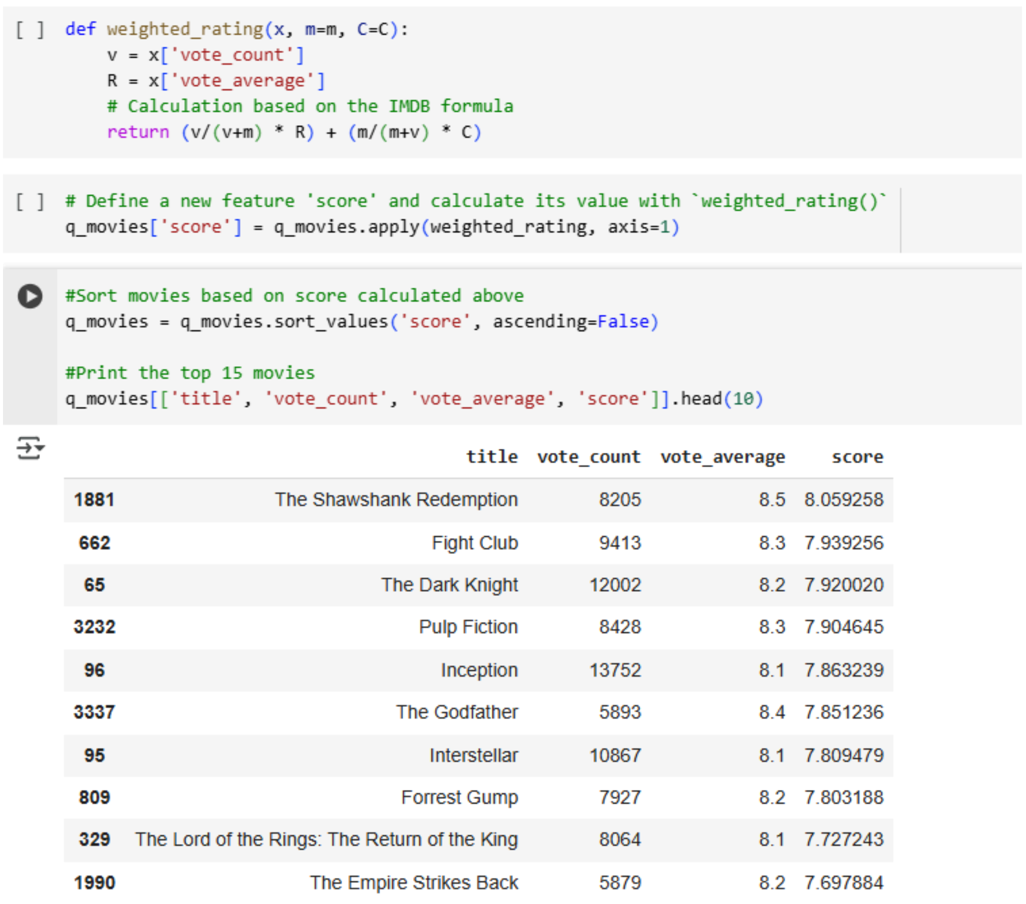
	budget	genres	homepage	id	keywords	original_language	original_title	overview	popularity	production_companies	...	runtime	spoken_languages	status	tagline	title	vote_average	vote_count	tittle	cast	crew
0	237000000	[{"id": 28, "name": "Action"}, {"id": 12, "nam...	http://www.avatarmovie.com/	19995	[{"id": 1463, "name": "culture clash"}, {"id":...	en	Avatar	In the 22nd century, a paraplegic Marine is di...	150.437577	[{"name": "Ingenious Film Partners", "id": 289...	...	162.0	[{"iso_639_1": "en", "name": "English"}, {"iso...	Released	Enter the World of Pandora.	Avatar	7.2	11800	Avatar	[{"cast_id": 242, "character": "Jake Sully", "...	[{"credit_id": "52fe48009251416c750aca23", "de...
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...	139.082615	[{"name": "Walt Disney Pictures", "id": 2}, {"...	...	169.0	[{"iso_639_1": "en", "name": "English"}]	Released	At the end of the world, the adventure begins.	Pirates of the Caribbean: At World's End	6.9	4500	Pirates of the Caribbean: At World's End	[{"cast_id": 4, "character": "Captain Jack Spa...	[{"credit_id": "52fe4232c3a36847f800b579", "de...
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...	107.376788	[{"name": "Columbia Pictures", "id": 5}, {"nam...	...	148.0	[{"iso_639_1": "fr", "name": "Fran\u00e7ais"},...	Released	A Plan No One Escapes	Spectre	6.3	4466	Spectre	[{"cast_id": 1, "character": "James Bond", "cr...	[{"credit_id": "54805967c3a36829b5002c41", "de...
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...	112.312950	[{"name": "Legendary Pictures", "id": 923}, {"...	...	165.0	[{"iso_639_1": "en", "name": "English"}]	Released	The Legend Ends	The Dark Knight Rises	7.6	9106	The Dark Knight Rises	[{"cast_id": 2, "character": "Bruce Wayne / Ba...	[{"credit_id": "52fe4781c3a36847f81398c3", "de...
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...	43.926995	[{"name": "Walt Disney Pictures", "id": 2}]	...	132.0	[{"iso_639_1": "en", "name": "English"}]	Released	Lost in our world, found in another.	John Carter	6.1	2124	John Carter	[{"cast_id": 5, "character": "John Carter", "c...	[{"credit_id": "52fe479ac3a36847f813eaa3", "de...

5 rows × 23 columns

Finally, we display a preview of the merged DataFrame to make sure everything is correct before moving on to the next steps. This helps confirm that the data is well-organized and ready for analysis or creating a model. By merging and preparing the data, we now have a complete dataset with all the important details, like movie titles, ratings, actors, and directors, which we can use in the following stages of the project.

B. Processing Popularity Data

Before using the data for recommendations, we need to organize and prepare it. Here, we calculate a "weighted rating" for each movie. This takes into account the number of votes it has received, the average rating, and a reference score from the overall dataset. The reason for this is to make sure movies are ranked by their true popularity, rather than just their average rating, which can be misleading if a movie has only a few ratings. Once we've done this, the data is ready to help us recommend popular movies or use it in other types of analysis.



```python
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)
```

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

```python
#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)
```

	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

In the code, the function **weighted_rating** calculates an adjusted score for each movie. It takes in a movie (x) and two numbers, **m** and **C**, which represent the minimum number of votes a movie must have to be considered, and the average rating of all movies, respectively.

Inside the function, **v** represents the number of votes for a movie, and **R** is its average rating. The formula is based on IMDb's ranking system, where the movie's rating **R** is combined with the overall average **C**, but the number of votes **v** is also considered. If a movie has a lot of votes, its rating matters more. If it has fewer votes, the overall average rating **C** plays a bigger role in adjusting the score.

Once the function is defined, we apply it to the dataset using: q_movies['score'] = q_movies.apply(weighted_rating, axis=1) This creates a new column called **score**, which contains the weighted rating of each movie.

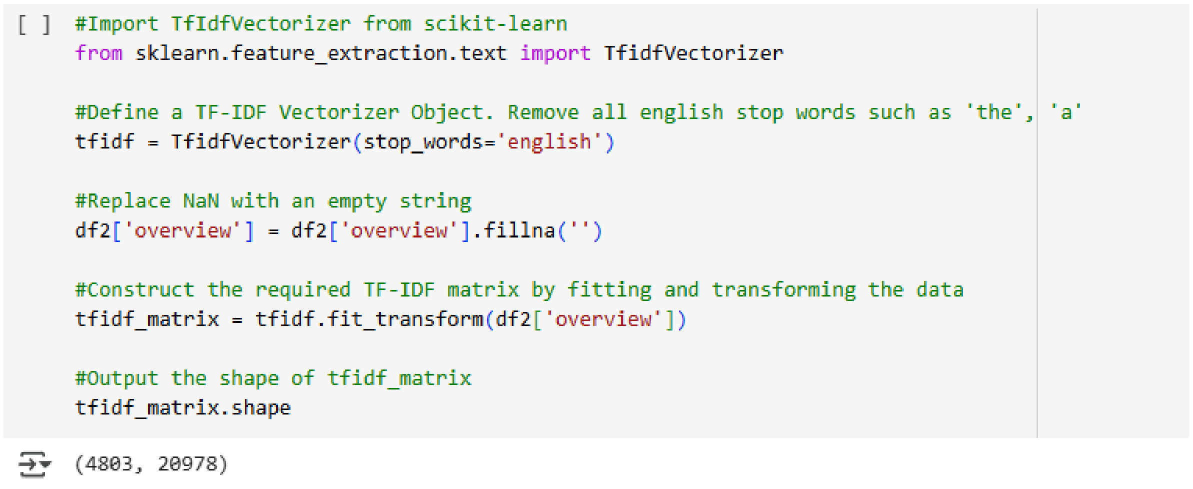
Next, we sort the movies by this score: q_movies = q_movies.sort_values('score', ascending=False) This provides a ranked list of movies from the most to the least popular.

Finally, we display the **top 10 movies** based on popularity: q_movies[['title', 'vote_count', 'vote_average', 'score']].head(10). This allows us to see the highest-rated movies based on their weighted popularity score.

This method helps us recommend movies based on their actual popularity, considering both ratings and the number of votes.

C. Content-Based Recommendation (Similarity)

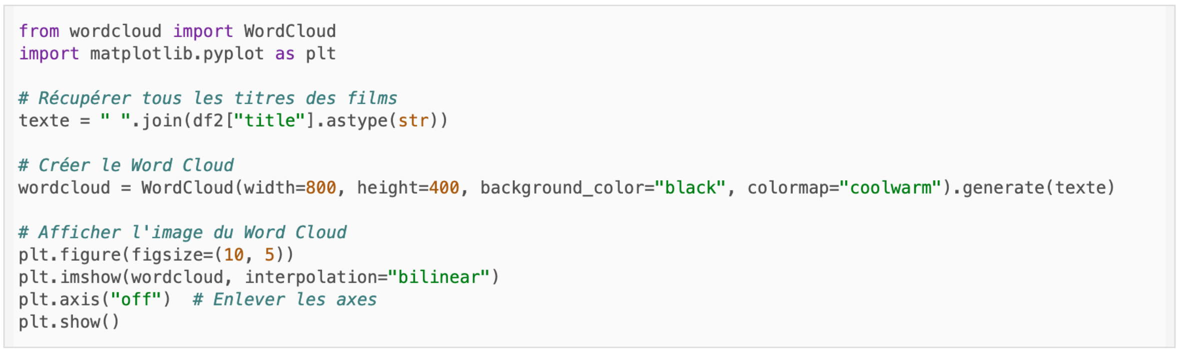
Content-based recommendation suggests movies based on their descriptions rather than their popularity. The idea is to analyze movie summaries to identify similar ones.



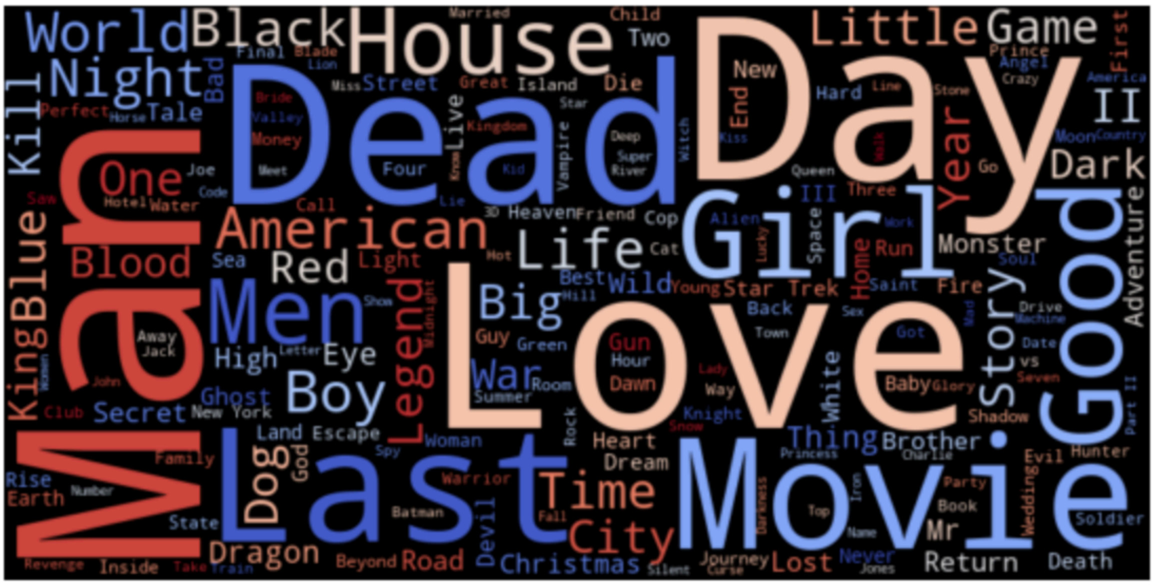
```
[ ] #Import TfIdfVectorizer from scikit-learn
    from sklearn.feature_extraction.text import TfidfVectorizer

    #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)
```

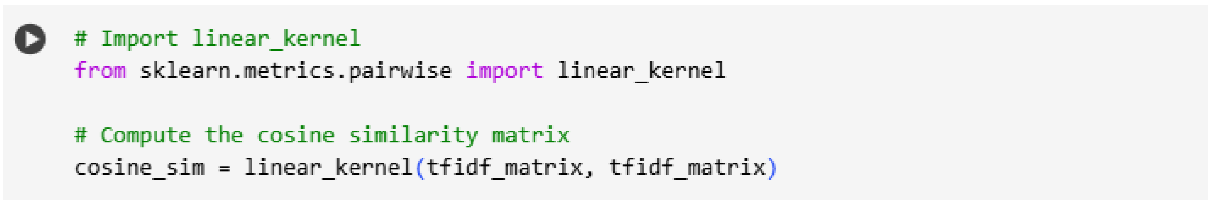
To achieve this, we first transform the **summaries into usable data** using **TF-IDF (Term Frequency-Inverse Document Frequency)**. This representation converts each movie into a **numerical vector**, emphasizing important words while reducing the influence of overly common words like "the" or "a."



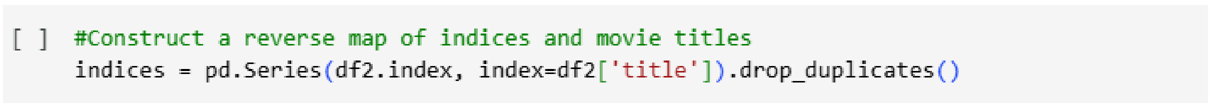
```
from wordcloud import WordCloud
import matplotlib.pyplot as plt

# Récupérer tous les titres des films
texte = " ".join(df2["title"].astype(str))

# Créer le Word Cloud
wordcloud = WordCloud(width=800, height=400, background_color="black", colormap="coolwarm").generate(texte)

# Afficher l'image du Word Cloud
plt.figure(figsize=(10, 5))
plt.imshow(wordcloud, interpolation="bilinear")
plt.axis("off")  # Enlever les axes
plt.show()
```



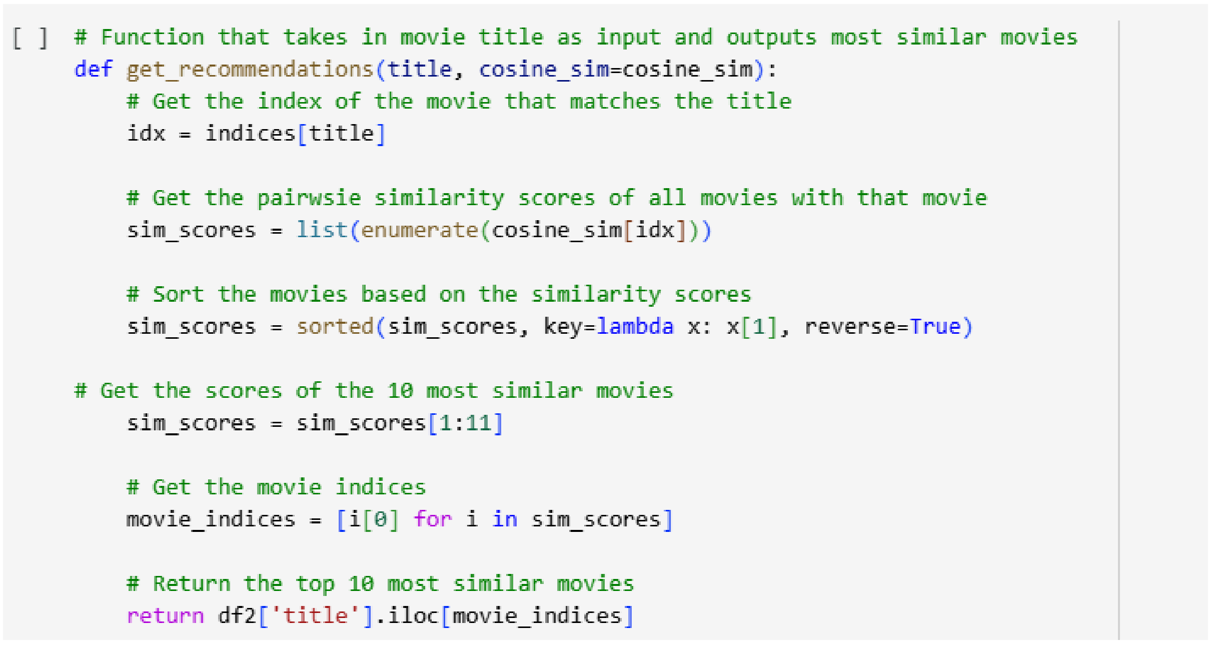
Once we had built the tf-idf we wanted to see the words in the synopsis that stood out the most. Here we can see that the words dead, love, last, america, stand out the most, fairly general terms but which signify the main themes of the 5000 films ranked on TMDB.



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

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

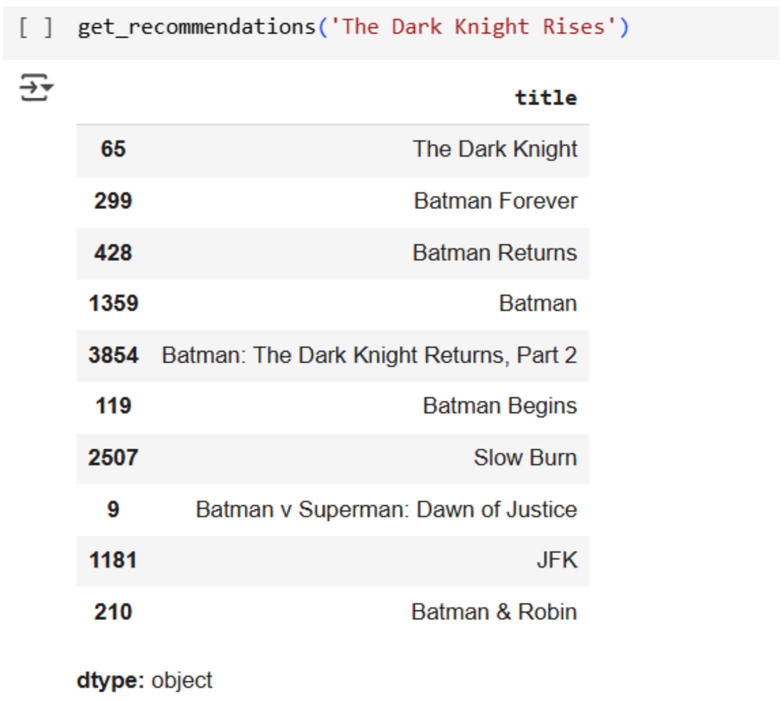


```python
#Construct a reverse map of indices and movie titles
indices = pd.Series(df2.index, index=df2['title']).drop_duplicates()
```



```python
# 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]
```

Once the summaries are converted into vectors, and we've done the world cloud, we calculate the **cosine similarity** between them. This measure evaluates how close two movies are by comparing their descriptions as numerical vectors. The higher the value, the more similar the movies.



```
[ ] get_recommendations('The Dark Knight Rises')
```

	title
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

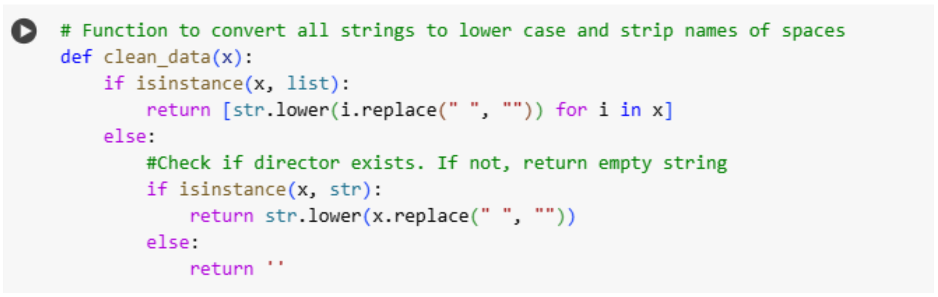
dtype: object

Using this **similarity matrix**, we can recommend movies similar to a given movie. We retrieve the most similar movies based on their similarity score and display the closest matches to the selected film.

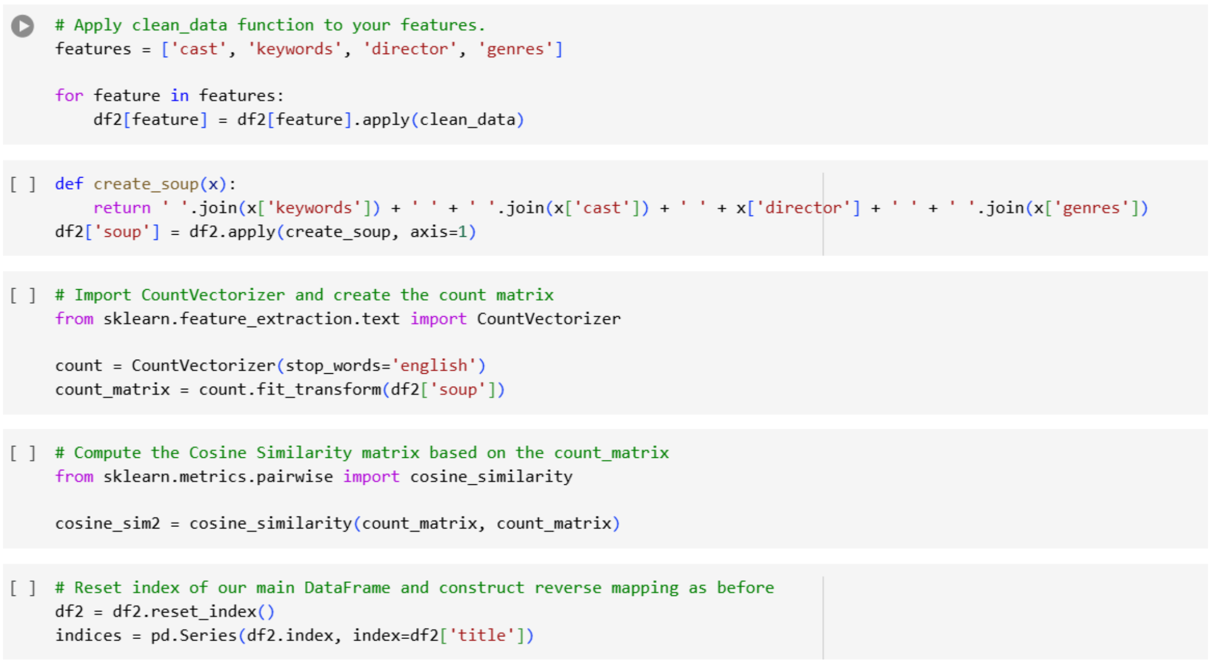
Thus, if a user likes a movie, this model will suggest other movies with **similar themes, stories, or genres**.

D. Enhanced Recommendations with Actors, Genres, and Directors

To improve movie recommendations, we can go beyond simple summaries by including additional information such as **actors, directors, genres, and associated keywords**. These details help capture each movie's unique characteristics, making recommendations more precise.



```python
# 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)
```

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

```
# 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)
```

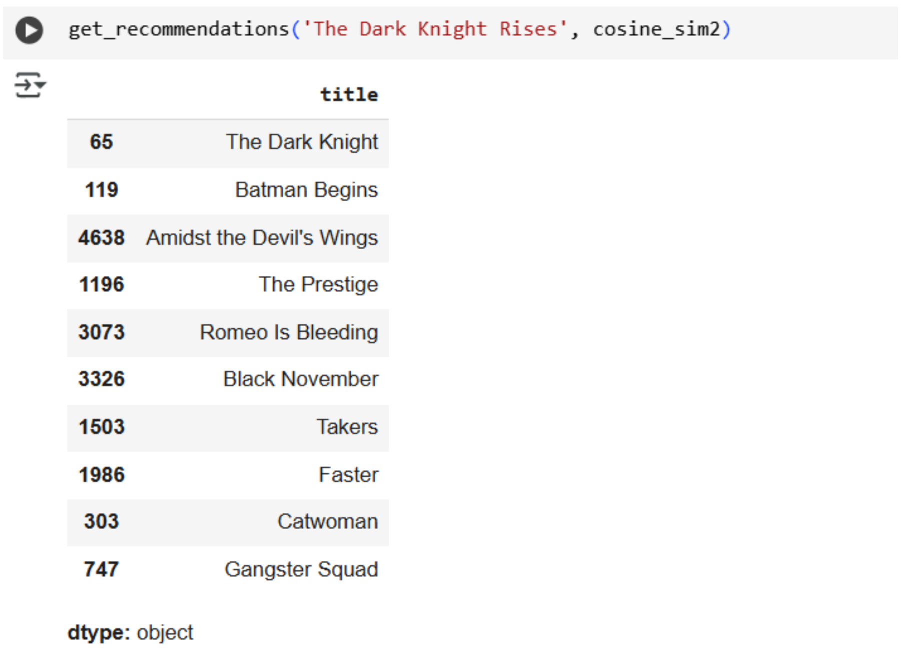
```
# Reset index of our main DataFrame and construct reverse mapping as before
df2 = df2.reset_index()
indices = pd.Series(df2.index, index=df2['title'])
```

The process begins with **data cleaning**. We apply a function that standardizes all names by converting them to **lowercase and removing unnecessary spaces**. This ensures consistency before using the data in computations. We apply this function to several columns: **actors, keywords, directors, and genres**.

Next, all this information is **combined into a single column** called **"soup."** This column contains a sequence of keywords, actors, directors, and genres for each movie. The idea is to **group all relevant information** that describes a movie comprehensively.

Once this step is complete, we use **CountVectorizer** to convert this text into a numerical matrix. **CountVectorizer** counts the occurrences of each word across movies, focusing on significant words while ignoring common stop words like "the" or "a."

From this matrix, we compute the **cosine similarity** between movies, allowing us to measure their similarity based on **combined characteristics (actors, director, genre, keywords).** The higher the similarity, the more alike the movies are.



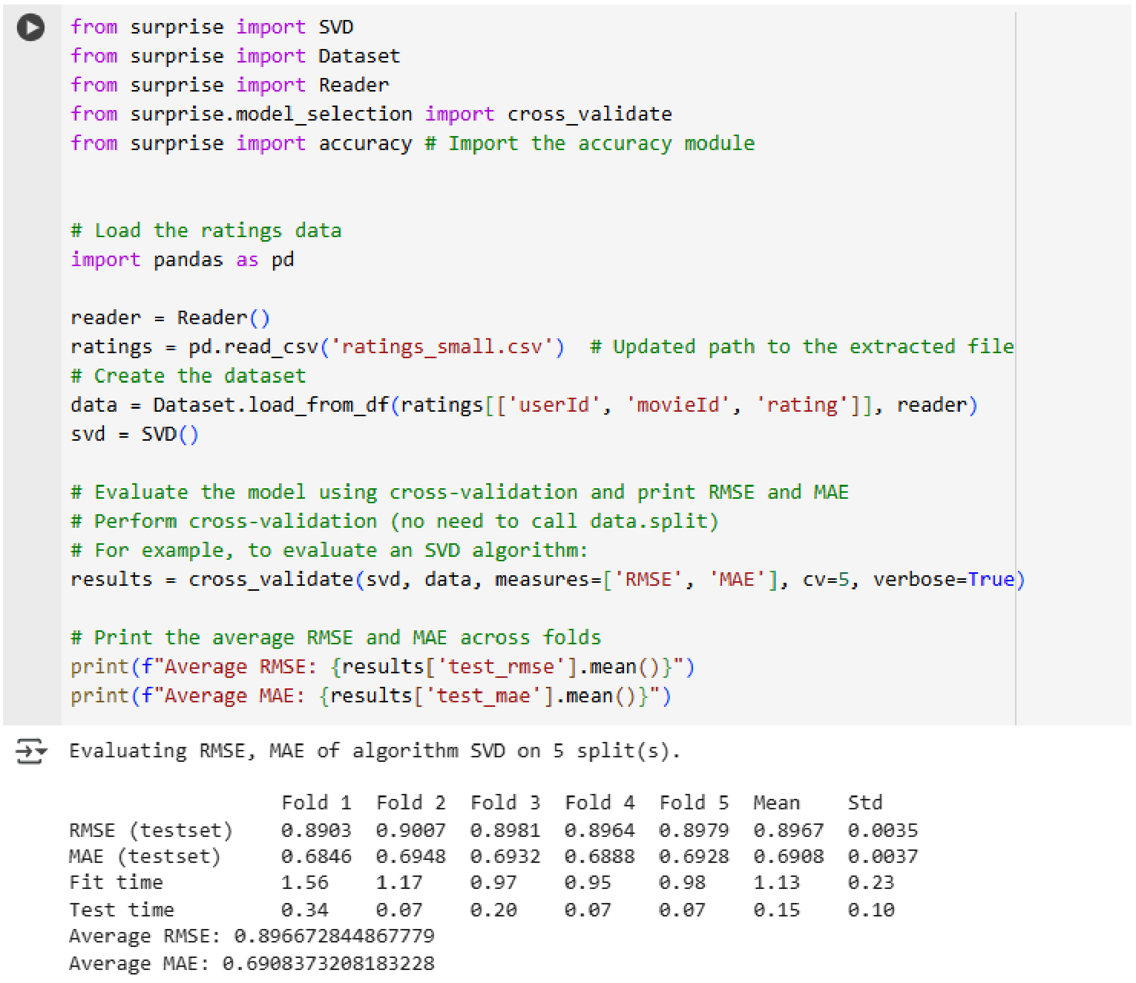
```
get_recommendations('The Dark Knight Rises', cosine_sim2)
```

```
                      title
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

dtype: object
```

Finally, we use this **similarity matrix** to recommend movies that are most similar to a user's preferred film. For example, if a user likes a particular movie, the model can suggest other films that share **similar actors, genres, or directors.** This results in **more personalized recommendations** that consider a broader range of factors.

E. Collaborative Recommendation with SVD Filtering

In this **collaborative filtering** approach, recommendations are based on past user ratings. We use a dataset containing user ratings for different movies, aiming to predict which movies each user might like but hasn't rated yet.

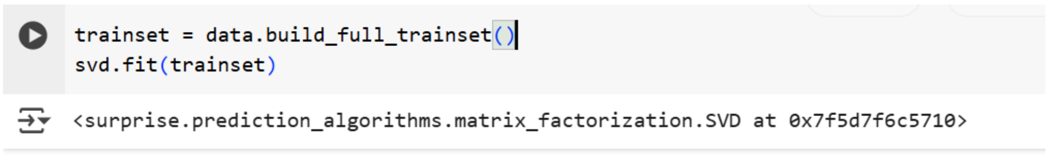


```python
from surprise import SVD
from surprise import Dataset
from surprise import Reader
from surprise.model_selection import cross_validate
from surprise import accuracy # Import the accuracy module


# Load the ratings data
import pandas as pd

reader = Reader()
ratings = pd.read_csv('ratings_small.csv')  # Updated path to the extracted file
# Create the dataset
data = Dataset.load_from_df(ratings[['userId', 'movieId', 'rating']], reader)
svd = SVD()

# Evaluate the model using cross-validation and print RMSE and MAE
# Perform cross-validation (no need to call data.split)
# For example, to evaluate an SVD algorithm:
results = cross_validate(svd, data, measures=['RMSE', 'MAE'], cv=5, verbose=True)

# Print the average RMSE and MAE across folds
print(f"Average RMSE: {results['test_rmse'].mean()}")
print(f"Average MAE: {results['test_mae'].mean()}")
```

```
Evaluating RMSE, MAE of algorithm SVD on 5 split(s).

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
RMSE (testset)    0.8903  0.9007  0.8981  0.8964  0.8979  0.8967  0.0035
MAE (testset)     0.6846  0.6948  0.6932  0.6888  0.6928  0.6908  0.0037
Fit time          1.56    1.17    0.97    0.95    0.98    1.13    0.23
Test time         0.34    0.07    0.20    0.07    0.07    0.15    0.10
Average RMSE: 0.896672844867779
Average MAE: 0.6908373208183228
```

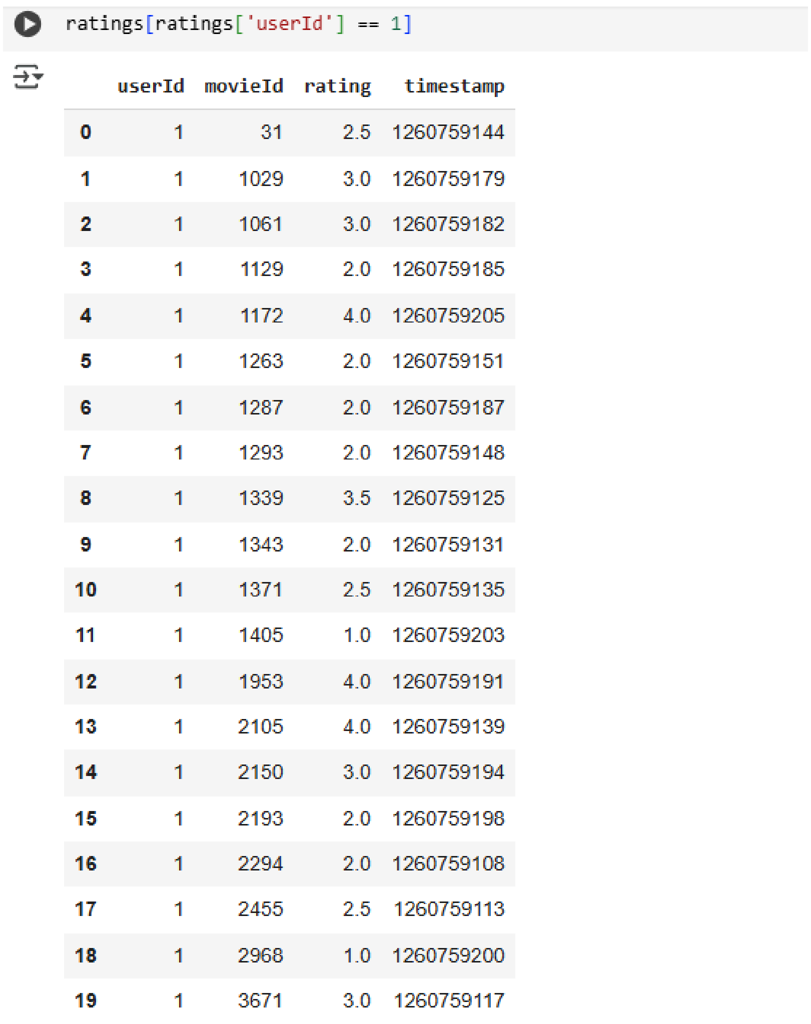
The **ratings_small.csv** file contains user ratings for various movies. We apply **Singular Value Decomposition (SVD)** to train a model that can predict missing ratings.

The process begins by **loading the data** using pandas and creating a **Dataset** with **Surprise**, a specialized library for recommendation systems. We then use an **SVD object** to build the model, and the **cross_validate** method to evaluate its performance using **RMSE (Root Mean Squared Error)** and **MAE (Mean Absolute Error)**. These metrics indicate the model's prediction accuracy on test data.



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

```
<surprise.prediction_algorithms.matrix_factorization.SVD at 0x7f5d7f6c5710>
```



```
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
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

After performing **cross-validation**, the model is trained on the entire dataset using fit(), allowing us to make predictions. For instance, the command: predict(1, 302, 3) predicts the **rating that user 1 would give to movie 302**.

This process enables us to **predict movies a user might enjoy** based on the past ratings of other users. Using **matrix decomposition via SVD**, the model identifies movies with similar rating patterns.

V. Conclusion

This notebook explores three **recommendation methods**:

1. **Popularity-based recommendations** – which Rely on global audience votes.
2. **Content-based recommendations** –it Identify movies with similar descriptions.
3. **Collaborative filtering (SVD)** –it Personalize recommendations based on user preferences.

Each method has its advantages:

- **Popularity-based filtering** is **simple and fast** but lacks personalization.
- **Content-based filtering** is useful for users looking for **similar movies**.
- **Collaborative filtering (SVD)** is the most advanced, as it **learns individual user preferences**.

By combining these approaches, we can create a **hybrid recommendation system** similar to those used by **Netflix or Amazon Prime**, improving recommendation accuracy and user experience.

Working on this project, we not only strengthened (strength ten) our machine learning skills but also developed a better understanding of the practical challenges in implementing such systems. This experience has been invaluable in teaching us how to handle data processing, model evaluation and system optimization, which are important when building scalable, user-centered recommendation systems.

Finally, this project has shown us how combining different recommendation strategies (such as content-based and collaborative filtering) can lead to more accurate and personalized suggestions by improving the overall (over rall) user experience. It has also helped us better understand the specific terms and techniques involved in running such a program.