Movie Recommendation System Using Python and Pandas

Project Overview

The goal of this project is to build a movie recommendation system using a dataset of movies and ratings of over 62,000 movies. The system suggests movies based on user preferences and similarity to other users. I utilized Python and the powerful pandas library to perform data manipulation and analysis, and the scikit-learn library for vectorization and similarity calculations.

Reading movies data

In [1]:
```python
import pandas as pd
movies = pd.read_csv("movies.csv")
```

In [2]:
```python
num_rows, num_cols = movies.shape
# Print the number of rows and columns
print("Number of rows:", num_rows)
print("Number of columns:", num_cols)
```

```
Number of rows: 62423
Number of columns: 3
```

In [3]:
```python
movies
```

Out[3]:

	movieId	title	genres
0	1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy
1	2	Jumanji (1995)	Adventure|Children|Fantasy
2	3	Grumpier Old Men (1995)	Comedy|Romance
3	4	Waiting to Exhale (1995)	Comedy|Drama|Romance
4	5	Father of the Bride Part II (1995)	Comedy
...	...	...	...
62418	209157	We (2018)	Drama
62419	209159	Window of the Soul (2001)	Documentary
62420	209163	Bad Poems (2018)	Comedy|Drama
62421	209169	A Girl Thing (2001)	(no genres listed)
62422	209171	Women of Devil's Island (1962)	Action|Adventure|Drama

62423 rows × 3 columns

Cleaning movie titles using regex

Remove special characters and symbols from movie titles. This step helped standardize the titles and make them suitable for further analysis.

In [4]:
```python
import re

def clean_title(title):
    return re.sub("[^a-zA-Z0-9 ]","",title)
```

In [5]:
```python
movies["clean_title"]=movies["title"].apply(clean_title)
```

In [6]:
```python
movies
```

Out[6]:

	movieId	title	genres	clean_title
0	1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy	Toy Story 1995
1	2	Jumanji (1995)	Adventure|Children|Fantasy	Jumanji 1995
2	3	Grumpier Old Men (1995)	Comedy|Romance	Grumpier Old Men 1995
3	4	Waiting to Exhale (1995)	Comedy|Drama|Romance	Waiting to Exhale 1995
4	5	Father of the Bride Part II (1995)	Comedy	Father of the Bride Part II 1995
...	...	...	...	...
62418	209157	We (2018)	Drama	We 2018
62419	209159	Window of the Soul (2001)	Documentary	Window of the Soul 2001
62420	209163	Bad Poems (2018)	Comedy|Drama	Bad Poems 2018
62421	209169	A Girl Thing (2001)	(no genres listed)	A Girl Thing 2001
62422	209171	Women of Devil's Island (1962)	Action|Adventure|Drama	Women of Devils Island 1962

62423 rows × 4 columns

Creating a Term Frequency - Inverse Document Frequency (TF-IDF) Matrix

To generate movie recommendations, I created a Term Frequency-Inverse Document Frequency (TF-IDF) matrix. This matrix represents the importance of each word in the movie titles relative to the entire dataset. I used the scikit-learn library's TfidfVectorizer to transform the cleaned titles into a numerical representation suitable for similarity calculations.

In [7]:
```python
from sklearn.feature_extraction.text import TfidfVectorizer
vectorizer = TfidfVectorizer(ngram_range=(1,2)) #this will search two terms together
tfidf = vectorizer.fit_transform(movies["clean_title"])
```

Creating a search function

This function takes a movie title as input, cleans it, and compares it to the TF-IDF matrix. The search function then calculates the cosine similarity between the input title and all movie titles in the dataset. It returns the top five most similar movies based on the search term.

In [8]:
```python
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np

def search(title):
    title = clean_title(title)
    query_vec = vectorizer.transform([title])
    similarity = cosine_similarity(query_vec, tfidf).flatten()
    #compare each of the clean titles to the query term and will return how similar it is
    indices = np.argpartition(similarity,-5)[-5:]
    #give the indices of five most similar titles to our search term
    results = movies.iloc[indices][::-1]
    return results
```

Building an interactive search box using Jupyter

To provide an interactive experience, I utilized Jupyter and the ipywidgets library to build an interactive search box. Users can enter a movie title, and the search results will be displayed dynamically. The search box triggers the search function, and the similar movies are displayed instantly.

In [9]:
```python
import ipywidgets as widgets
from IPython.display import display

movie_input = widgets.Text(
    description = "Movie Title",
    disabled = False)

movie_list = widgets.Output() #output widget

def on_type(data):
    with movie_list:
        movie_list.clear_output() #first clear the output
        title = data["new"] #get the new title
        if len(title) > 5 :
            display(search(title))

movie_input.observe(on_type, names='value')
#when an input is received the on_type function is called , the observed event is of type value

display(movie_input,movie_list)
```

```
Text(value='Toy Story', description='Movie Title')
Output()
```

Reading movie ratings data

In [10]:
```python
ratings = pd.read_csv("ratings.csv")
```

In [11]:
```python
ratings
```

Out[11]:

	userId	movieId	rating	timestamp
0	1	296	5.0	1147880044
1	1	306	3.5	1147868817
2	1	307	5.0	1147868828
3	1	665	5.0	1147878820
4	1	899	3.5	1147868510
...	...	...	...	...
25000090	162541	50872	4.5	1240953372
25000091	162541	55768	2.5	1240951998
25000092	162541	56176	2.0	1240950697
25000093	162541	58559	4.0	1240953434
25000094	162541	63876	5.0	1240952515

25000095 rows × 4 columns

Finding Similar Users

To personalize recommendations further, I analyzed movie ratings data to find users who liked the same movie. By filtering the ratings dataset based on movie preferences, I identified similar users who rated a particular movie highly.

In [12]:
```python
ratings.dtypes
```

Out[12]:
```
userId         int64
movieId        int64
rating       float64
timestamp      int64
dtype: object
```

In [13]:
```python
movie_id=89745
```

In [14]:
```python
similar_users = ratings[(ratings["movieId"]==movie_id) & (ratings["rating"] >4)]["userId"].unique()
```

In [15]:
```python
similar_users
```

Out[15]:
```
array([    21,    187,    208, ..., 162469, 162485, 162532], dtype=int64)
```

Recommending Similar Movies

Next, I identified other movies that the similar users liked. I retrieved the movies that were rated highly by the identified users. This step helped in finding movies with similar interests to the initial movie choice.

In [16]:
```python
similar_user_recs = ratings[(ratings["userId"].isin(similar_users)) & (ratings["rating"] >4)]["movieId"]
similar_user_recs
```

Out[16]:
```
3741           318
3742           527
3743           541
3744           589
3745           741
             ...
24998517     91542
24998518     92259
24998522     98809
24998523    102125
24998524    112852
Name: movieId, Length: 577796, dtype: int64
```

Finding movies liked by similar users

To narrow down the recommendations, I determined the movies that were liked by more than 10% of the similar users. By calculating the percentage of similar users who liked each movie, I filtered out movies with lower user affinity.

In [17]:
```python
similar_user_recs = similar_user_recs.value_counts() / len(similar_users)
```

In [18]:
```python
similar_user_recs = similar_user_recs[similar_user_recs > 0.10]
```

In [19]:
```python
similar_user_recs
```

Out[19]:
```
89745    1.000000
58559    0.573393
59315    0.530649
79132    0.519715
2571     0.496687
           ...
47610    0.103545
780      0.103380
88744    0.103048
1258     0.101226
1193     0.100895
Name: movieId, Length: 193, dtype: float64
```

Evaluating General User Preference

To assess the popularity of the recommended movies, I examined the ratings of all users. I identified the movies that had high ratings from the general user population.

In [20]:
```python
all_users = ratings[(ratings["movieId"].isin(similar_user_recs.index)) & (ratings["rating"]>4)]
```

In [21]:
```python
all_users
```

Out[21]:

	userId	movieId	rating	timestamp
0	1	296	5.0	1147880044
29	1	4973	4.5	1147869080
48	1	7361	5.0	1147880055
72	2	110	5.0	1141416589
76	2	260	5.0	1141417172
...	...	...	...	...
25000065	162541	5952	5.0	1240952617
25000078	162541	7153	5.0	1240952613
25000081	162541	7361	4.5	1240953484
25000086	162541	31658	4.5	1240953287
25000090	162541	50872	4.5	1240953372

1893092 rows × 4 columns

In [22]:
```python
#finding the percentage of all users who like the similar movies recommended
#there should be a differential betweent the % of users who have a similar interest like you vs the general set of users
all_user_recs = all_users["movieId"].value_counts()/ len(all_users["userId"].unique())
```

In [23]:
```python
all_user_recs
```

Out[23]:
```
318       0.346395
296       0.288146
2571      0.247010
356       0.238136
593       0.228665
            ...
86332     0.010142
91630     0.009324
122900    0.008573
122926    0.008070
106072    0.005289
Name: movieId, Length: 193, dtype: float64
```

Creating a recommendation Score

To quantify the recommendation quality, I calculated a recommendation score for each movie. This score represented the ratio of similar users who liked a movie to the overall users who liked the movie. A higher score indicated a better recommendation.

In [24]:
```python
rec_percentages = pd.concat([similar_user_recs,all_user_recs],axis=1)
rec_percentages.columns = ["similar","all"]
```

In [25]:
```python
rec_percentages #this wll show the comparison between the ratings of similar users and general users
```

Out[25]:

	similar	all
89745	1.000000	0.040459
58559	0.573393	0.148256
59315	0.530649	0.054931
79132	0.519715	0.132987
2571	0.496687	0.247010
...	...	...
47610	0.103545	0.022770
780	0.103380	0.054723
88744	0.103048	0.010383
1258	0.101226	0.083887
1193	0.100895	0.120244

193 rows × 2 columns

In [26]:
```python
rec_percentages["score"]= rec_percentages["similar"]/ rec_percentages["all"]
```

In [27]:
```python
rec_percentages = rec_percentages.sort_values("score",ascending=False)
rec_percentages
```

Out[27]:

	similar	all	score
89745	1.000000	0.040459	24.716368
106072	0.103711	0.005289	19.610199
122892	0.241054	0.012367	19.491770
102125	0.216534	0.012119	17.867419
88140	0.215043	0.012052	17.843074
...	...	...	...
296	0.288933	0.288146	1.002730
593	0.222830	0.228665	0.974483
527	0.199967	0.217833	0.917984
1193	0.100895	0.120244	0.839081
2858	0.139993	0.169679	0.825051

193 rows × 3 columns

The higher the score the better the recommendation.

In [28]:
```python
rec_percentages.head(10).merge(movies, left_index = True, right_on = "movieId")
```

Out[28]:

	similar	all	score	movieId	title	genres	clean_title
17067	1.000000	0.040459	24.716368	89745	Avengers, The (2012)	Action|Adventure|Sci-Fi|IMAX	Avengers The 2012
20513	0.103711	0.005289	19.610199	106072	Thor: The Dark World (2013)	Action|Adventure|Fantasy|IMAX	Thor The Dark World 2013
25058	0.241054	0.012367	19.491770	122892	Avengers: Age of Ultron (2015)	Action|Adventure|Sci-Fi	Avengers Age of Ultron 2015
19678	0.216534	0.012119	17.867419	102125	Iron Man 3 (2013)	Action|Sci-Fi|Thriller|IMAX	Iron Man 3 2013
16725	0.215043	0.012052	17.843074	88140	Captain America: The First Avenger (2011)	Action|Adventure|Sci-Fi|Thriller|War	Captain America The First Avenger 2011
16312	0.175447	0.010142	17.299824	86332	Thor (2011)	Action|Adventure|Drama|Fantasy|IMAX	Thor 2011
21348	0.287608	0.016737	17.183667	110102	Captain America: The Winter Soldier (2014)	Action|Adventure|Sci-Fi|IMAX	Captain America The Winter Soldier 2014
25071	0.214049	0.012856	16.649399	122920	Captain America: Civil War (2016)	Action|Sci-Fi|Thriller	Captain America Civil War 2016
25061	0.136017	0.008573	15.865628	122900	Ant-Man (2015)	Action|Adventure|Sci-Fi	AntMan 2015
14628	0.242876	0.015517	15.651921	77561	Iron Man 2 (2010)	Action|Adventure|Sci-Fi|Thriller|IMAX	Iron Man 2 2010

Building a recommendation Function

I encapsulated the recommendation logic into a function. This function takes a movie ID as input and returns the top ten recommended movies based on the identified similar users' preferences. The function calculates the recommendation score and sorts the movies accordingly.

In [33]:
```python
def find_similar_movies(movie_id):
    similar_users = ratings[(ratings["movieId"] == movie_id) & (ratings["rating"] > 3)]["userId"].unique()
    similar_user_recs = ratings[(ratings["userId"].isin(similar_users)) & (ratings["rating"] > 3)]["movieId"]
    similar_user_recs = similar_user_recs.value_counts() / len(similar_users)

    similar_user_recs = similar_user_recs[similar_user_recs > .10]
    all_users = ratings[(ratings["movieId"].isin(similar_user_recs.index)) & (ratings["rating"] > 3)]
    all_user_recs = all_users["movieId"].value_counts() / len(all_users["userId"].unique())
    rec_percentages = pd.concat([similar_user_recs, all_user_recs], axis=1)
    rec_percentages.columns = ["similar", "all"]

    rec_percentages["score"] = rec_percentages["similar"] / rec_percentages["all"]
    rec_percentages = rec_percentages.sort_values("score", ascending=False)
    return rec_percentages.head(10).merge(movies, left_index=True, right_on="movieId")[["score", "title", "genres"]]
```

Creating an interactive recommendation widget

To provide an interactive and user-friendly interface, I created a recommendation widget. Users can enter the title of a movie they enjoyed, and the widget dynamically displays the top recommendations. It utilizes the search function to find the closest matching movie and then recommends similar movies based on the identified similar users.

In [34]:
```python
movie_name_input = widgets.Text(
    description='Movie Title:',
    disabled=False
)
recommendation_list = widgets.Output()

def on_type(data):
    with recommendation_list:
        recommendation_list.clear_output()
        title = data["new"]
        if len(title) > 5:
            results = search(title)
            movie_id = results.iloc[0]["movieId"]
            display(find_similar_movies(movie_id))

movie_name_input.observe(on_type, names='value')

display(movie_name_input, recommendation_list)
```

```
Text(value='', description='Movie Title:')
Output()
```