**Introduction**

In this project, we have focused on implementing a content-based movie recommendation system. The content-based recommendation system suggests movies to users based on items they have previously liked or their specific profile characteristics. Essentially, this type of system assumes that if a user has shown interest in a particular item in the past, they are likely to be interested in similar items now. Similar items are grouped together, and user profiles are built based on their purchase history or by explicitly asking the user about their preferences through a series of questions. There are also methods that use personal user information and data from social networks, although they are not purely content-focused. In summary, in the content-based model, items are grouped, and if a user likes an item from a particular group, other items from the same group are recommended to them.

In this project, we have used two datasets available on GitHub in CSV format. A CSV (Comma-Separated Values) file is a file containing text separated by commas but can also be separated by any other character. After merging these two datasets, data preprocessing operations were performed, including finding missing data, handling duplicates, and cleaning the data. We selected features that we would use to identify similarities between movies and proceeded to implement the model. Below, we will explain the project's stages in more detail:

**Dataset**

As mentioned earlier, in this project, we used two datasets. The first dataset is named "movies," and the second dataset is named "credits." The "movies" dataset contains 5000 rows or movies with 19 features.

**Features:**

1. Budget

2. Genre

3. Homepage

4. ID

5. Keywords

6. Original Language

7. Overview

8. Popularity

9. Production Companies

10. Production Countries

11. Release Date

12. Revenue

13. Runtime

14. Status

15. Tagline

16. Vote Average

17. Vote Count

18. Spoken Languages

19. Title

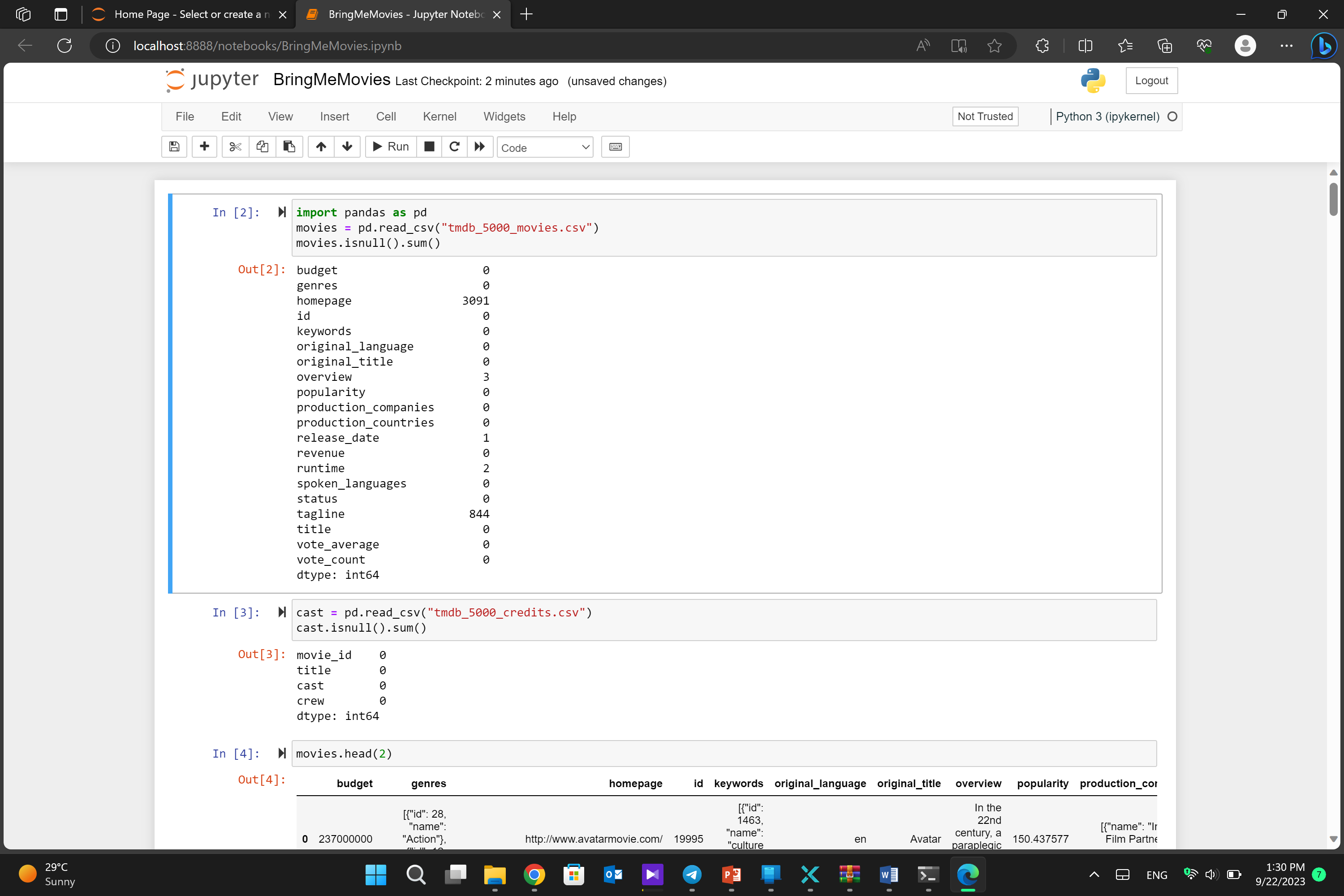
**Data Preprocessing**

**Importance of Data Preprocessing**

In the field of data mining, all data that will be used for the intended purpose must be prepared and adjusted using methods before starting processing, known as "preprocessing." Data preparation before processing is called preprocessing, and it plays a fundamental role in the data processing process and the results obtained from it. Since data is usually obtained from sources that have generated or stored the data without considering data mining processes, it is necessary to prepare the data according to the conditions and the problem, making it suitable for injection into data mining algorithms. In the following, we will explain the data preprocessing steps used in the project.

**1. Missing Data**

First, we read the first dataset using the powerful pandas library and then use the `isnull` and `sum` commands to find the number of missing data points. The `isnull` command returns true for values that are null and false for others. Since a true value is considered as 1 and a false value is considered as 0, summing these values helps us determine the number of missing data points for each feature. This function, by default, goes through the columns and reports the count of missing data for each feature. Below is the code and output for this part of the code:

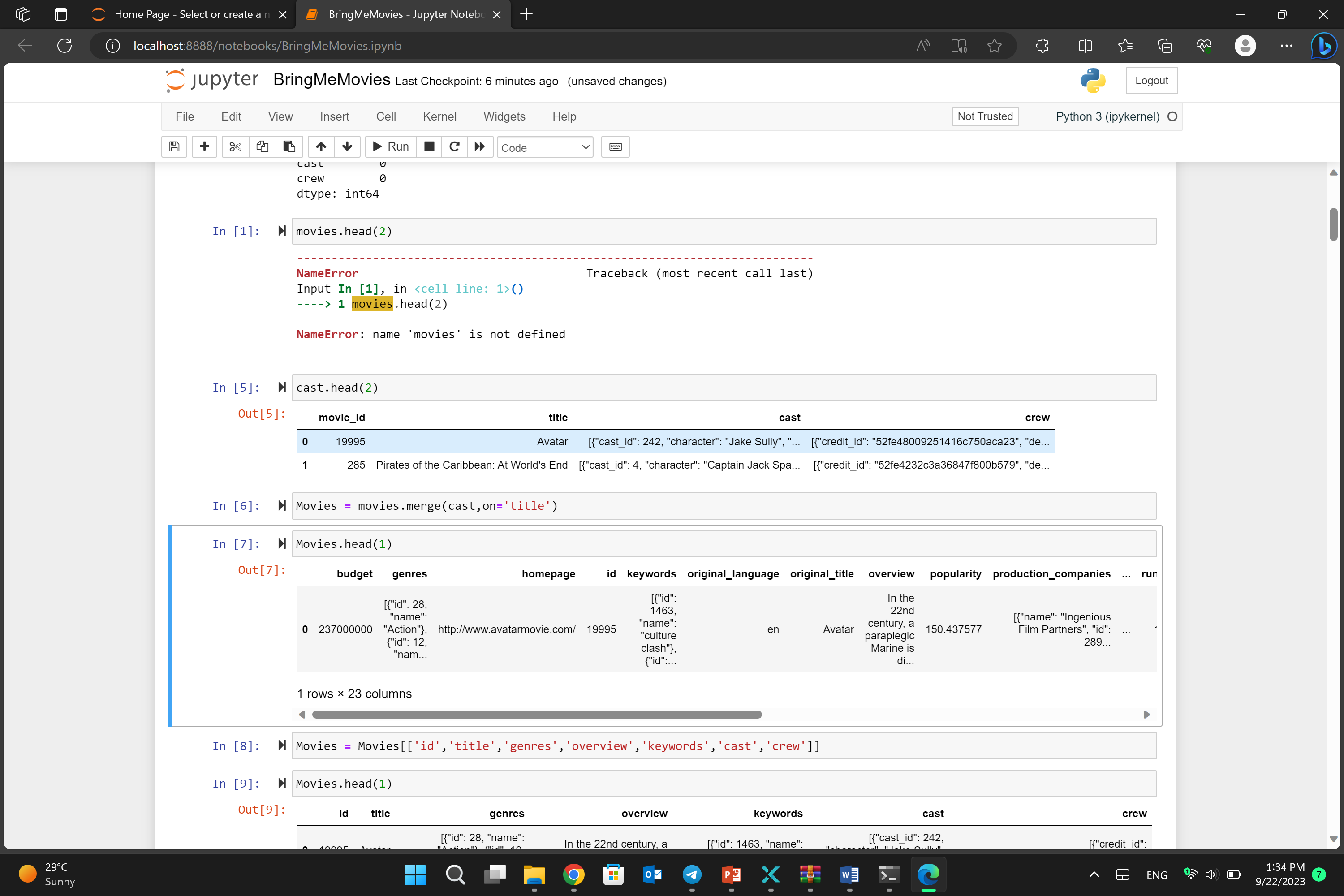


As seen in the output, for example, the "homepage" feature has 3091 missing data points, and the "tagline" feature has 844 missing data points.

The same process is performed for the "credit" file. The data is read, stored in a variable called "cast”

**2. Merging Datasets**

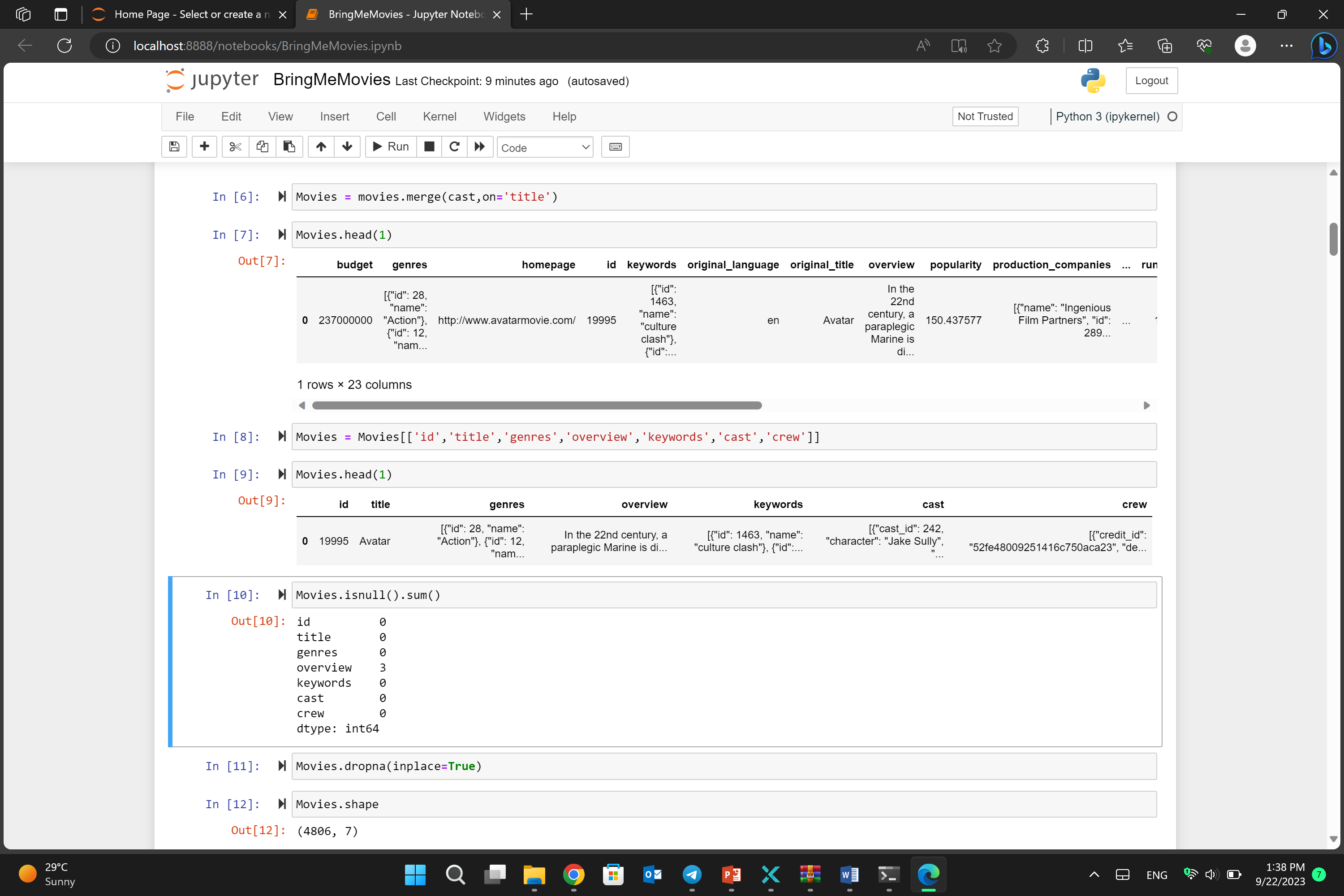
After reading the datasets, the next step is to merge or consolidate the datasets. This is done using the following command:



The above command merges the "cast" dataset with the "movies" dataset. Since both datasets have a common feature, "title," the `on` parameter is used to specify the common column. After merging, the data is stored in the "Movies" variable for further processing.

**3. Feature Selection**

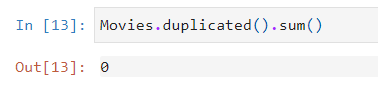
After merging the datasets, we need to select the features that we want to use to identify similarities between movies. Using a logical approach, we can see that the following features are suitable for our task. We select only these columns and place them in the "Movies" dataset:



The output of this code shows that there are three missing data points in the "overview" feature. So far, we have only found the count of missing data points, but considering that the "Movies" dataset is our final dataset, the next command in the code removes the missing data points within this dataset. Therefore, this command does not return a new dataset but performs the data cleaning operation on the original dataset.

**4. Duplicate Data**

Duplicate data can lead to an increase in the volume of data being used and make working with data more challenging due to redundancy and decreased readability. To identify and handle duplicate data, the following command is used:



As observed, there is no duplicate data; therefore, we can proceed with the implementation using this dataset.

**Creating a Genres List**

The "Genres" column is structured as follows. Since we only need a list of film genres, we need to perform a cleaning operation on this column. The first row of the "Genres" column appears as follows:

'[{"id": 28, "name": "Action"}, {"id": 12, "name": "Adventure"}, {"id": 14, "name": "Fantasy"}, {"id": 878, "name": "Science Fiction"}]'

To clean this data, we define a function called "CreateList." Initially, an empty list is created, and with the help of JSON parsing, wherever we reach the "name" part, its value is added to the list.

import json

def CreateList(obj):

NewList = []

for i in json.loads(obj):

NewList.append(i['name'])

return NewList

**Creating Keyword List**

The "keywords" column before cleansing is as follows:

'[{"id": 1463, "name": "culture clash"}, {"id": 2964, "name": "future"}, {"id": 3386, "name": "space war"}, {"id": 3388, "name": "space colony"}, {"id": 3679, "name": "society"}, {"id": 3801, "name": "space travel"}, {"id": 9685, "name": "futuristic"}, {"id": 9840, "name": "romance"}, {"id": 9882, "name": "space"}, {"id": 9951, "name": "alien"}, {"id": 10148, "name": "tribe"}, {"id": 10158, "name": "alien planet"}, {"id": 10987, "name": "cgi"}, {"id": 11399, "name": "marine"}, {"id": 13065, "name": "soldier"}, {"id": 14643, "name": "battle"}, {"id": 14720, "name": "love affair"}, {"id": 165431, "name": "anti war"}, {"id": 193554, "name": "power relations"}, {"id": 206690, "name": "mind and soul"}, {"id": 209714, "name": "3d"}]'

Keyword column after cleansing :

['culture clash', 'future', 'space war', 'space colony', 'society', 'space travel', 'futuristic', 'romance', 'space', 'alien', 'tribe', 'alien planet', 'cgi', 'marine', 'soldier', 'battle', 'love affair', 'anti war', 'power relations', 'mind and sol,'3d']

**Creating Cast List**

After processing the 'keywords' column in the same way, it's now time to address the 'Cast' column. Like other columns, the 'Cast' column also requires cleaning. We aim to convert the 'Cast' data into a list of the top 5 main actors/actresses for each film. To achieve this, we need to define a new function called 'CreateCastList.'

The 'CreateCastList' function is designed to extract the top 5 cast members from the JSON-formatted data in the 'Cast' column. Here's the code for the 'CreateCastList' function:

def CreateCastList(obj):

CastList = []

loop = 0

for cast in json.loads(obj):

if loop != 5:

CastList.append(cast['name'])

loop+=1

else:

break

return CastList

**Creating Crew List**

The "Crew" column, like the other columns, also requires cleaning. In this column, the names of all individuals involved in the film's production are provided along with their respective roles. To clean the "Crew" column, we create a new function called "CreateCrewList" as follows:

def CreateCrewList(obj):

CrewList = []

for crew in json.loads(obj):

if crew['job'] == 'Director':

CrewList.append(crew['name'])

return CrewList

In the "CreateCrewList" function, similar to the previous functions, we start by initializing an empty list called "CrewList." We then iterate through the crew members in the JSON data. We specifically add the names of directors to the list. For example, the output of this function for the movie "Avatar" would be:

['James Cameron']

In the next step, we apply all the functions we defined to the dataset's columns:

Movies['genres'] = Movies['genres'].apply(CreateList)

Movies['keywords'] = Movies['keywords'].apply(CreateList)

Movies['cast'] = Movies['cast'].apply(CreateCastList)

Movies['crew'] = Movies['crew'].apply(CreateCrewList)

Additionally, we remove any extra spaces as follows. If these spaces are not removed, they may be treated as differences between two films in subsequent analysis steps:

Movies['genres'] = Movies['genres'].apply(lambda x: [i.replace(' ','') for i in x])

Movies['cast'] = Movies['cast'].apply(lambda x:[i.replace(' ','') for i in x])

Movies['crew'] = Movies['crew'].apply(lambda x:[i.replace(' ','') for i in x ])

To facilitate comparisons between films, we can combine all five columns into one column. The "tags" column includes data from the five columns: Cast, Crew, Overview, Genres, and Keywords:

Movies['tags'] = Movies['overview'] + Movies['genres'] + Movies['keywords'] + Movies['cast'] + Movies['crew']

**Data Preprocessing**

Now, we have a DataFrame called "NewMovies" with three columns: Id, Title, and Tags. To provide recommendations for similar films, we first need to calculate the similarity between sentences in the "Tags" column. Before processing the "Tags" column, we need to make a series of changes:

1-Remove extra spaces to avoid issues in subsequent processing steps.

NewMovies['tags'] = NewMovies['tags'].apply(lambda x: ' '.join(x))

2-Convert all letters in words to lowercase (e.g., "G" to "g"). This step ensures that words have a consistent format within each sentence.

NewMovies['tags'] = NewMovies['tags'].apply(lambda x: x.lower())

Assume that in one sentence the word is "Become," in another sentence the word is "Becomes," and in yet another sentence, the word is "Becoming." These three words convey the same meaning, but a computer cannot recognize this and treats them as three different words. Therefore, we need to remove word suffixes and only use the root of the words (e.g., "Become" becomes "Becom"). To do this, we use the Porter Stemmer function.

from nltk.stem.porter import PorterStemmer

ps = PorterStemmer()

We define a function that takes a text as input and removes word suffixes from it, retaining only the word roots.

def CreateStem(text):

    StemList = []

    for word in text.split():

        StemList.append(ps.stem(word))

    return ' '.join(StemList)

Apply the "CreateStem" function to all sentences in the "Tags" column.

NewMovies['tags'] = NewMovies['tags'].apply(CreateStem)

These preprocessing steps help ensure that the text data is in a suitable format for further analysis, such as calculating similarity between sentences.

**Data Modeling**

One of the fundamental limitations of computers is that they only understand numerical data and do not comprehend the meaning of words and sentences. Film titles, people's names, and sentences are not inherently understood by computers. Therefore, to calculate the similarity between sentences, we need to transform them into a numerical representation. One such method is using the CountVectorizer function. This function works by considering all the words used in all sentences as attributes and then examining how many times each word appears in each sentence. It assigns a numerical value to each word based on its frequency in the sentence. For example, if the word "hello" appears twice in a sentence, it assigns the number 2, and if it doesn't appear at all, it assigns the number 0. The output is a matrix of size N × M, where N is the number of sentences, and M is the number of unique words. Each element is an M-dimensional array.

from sklearn.feature\_extraction.text import CountVectorizer

cv = CountVectorizer(max\_features=5000,stop\_words='english')

MoviesVectors = cv.fit\_transform(NewMovies['tags']).toarray()

Now that we have converted textual data into numerical data, we can determine the similarity between these vectors and, consequently, the similarity between the respective films. We use the Cosine Similarity method for calculating similarity. This method emphasizes that the closer two vectors are to each other, the more similar they are. The angle θ between them approaches zero, and as θ approaches zero, Cos θ approaches 1. This variable is calculated as follows:

The output is an N × N matrix, where N is the number of sentences. It computes the similarity of each sentence with all other sentences.

from sklearn.metrics.pairwise import cosine\_similarity

similarity =  cosine\_similarity(MoviesVectors)

Now, after calculating the similarity levels, we can define a function that, given the name of a film as input, returns the 5 films with the highest similarity. In essence, we provide the film's name to the function, the function finds the index of this film in the NewMovies DataFrame, checks the similarity list of this film using that index, sorts the list in descending order, and returns the top 5 films with the highest similarity.

def BringMeMovies(movie):

    movie\_index = NewMovies[NewMovies['title'] == movie].index[0]

    distances = similarity[movie\_index]

    SimilarMoviesList = sorted(list(enumerate(distances)),reverse = True , key = lambda x:x[1])[1:6]

    for similarMovies in SimilarMoviesList:

        print(NewMovies.iloc[similarMovies[0]].title)

You can run this function for a specific film, like 'Inception,' to get movie recommendations based on similarity. The output will be a list of recommended movies:

Abduction

12 Rounds

Rockaway

Memento

Hesher