REPORT

Name - Adarsh Kumar
Email - kumar.241@iitj.ac.in

Movie Recommender System

DATASET
https://www.kaggle.com/datasets/tmdb/tmdb-movie-metadata?select=tmdb_5000_movies.csv

Link to try it out
https://movierecommender-s8s7yjcx71.streamlit.app/

Colab File :
https://colab.research.google.com/drive/1_rgrwt1haCsC0_Eg2FPwWf9TnAEClvzW?usp=sharing

Summary : Type or Select name from the drop down menu and click on recommend , Top 5 similar movies related to your typed or selected movie will get displayed.

Work Flow Explanation

We have two dataset 'tmdb_5000_movies.csv' and 'tmdb_5000_credits.csv'
Where tmdb_5000_movies.csv has

budget	genres	homepage	id	keywords	original_language	original_title	overview	popularity	production_companies	production_countries
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...	[{"iso_3166_1": "US", "name": "United States o...

release_date	revenue	runtime	spoken_languages	status	tagline	title	vote_average	vote_count
2009-12-10	2787965087	162.0	[{"iso_639_1": "en", "name": "English"}, {"iso...	Released	Enter the World of Pandora.	Avatar	7.2	11800

And tmdb_5000_credits.csv has

```
credits.head(1)
```

	movie_id	title	cast	crew
0	19995	Avatar	[{"cast_id": 242, "character": "Jake Sully", "...	[{"credit_id": "52fe48009251416c750aca23", "de...

Then we merged both data frames

Features which we are going to include for our recommender system:

genres
id
keywords
title
overview
cast - (for actor or actress)
crew -(for director)

movies = movies[['genres' ,'movie_id' , 'keywords' , 'title' , 'overview' , 'cast' , 'crew']]

Then we found that the overview row has some null values , so we dropped them .
Now , 'genres' have
[{"id": 28, "name": "Action"}, {"id": 12, "name": "Adventure"}, {"id": 14, "name": "Fantasy"}, {"id": 878, "name": "Science Fiction"}]
We have to convert it to
['Action' , 'Adventure' , 'Fantasy' , 'Science Fiction']

So , we used The ast.literal_eval **method , it is one of the helper functions that helps traverse an abstract syntax tree.**

So , we made a convert function , which is as follows :

```
def convert(obj):
  l=[]
  for i in ast.literal_eval(obj):
    l.append(i['name'])
  return l
```

Then :
movies['genres']= movies['genres'].apply(convert)

Same we did with keyword column

Now , from cast column we include only top 3 cast as rest cast will be less known by people and will not contribute to our recommendation

So , for this we used another convert function :

```
def convert3(obj):
  l=[]
  counter=0
  for i in ast.literal_eval(obj):
    if counter !=3:
      l.append(i['name'])
      counter += 1
  return l
```

Now from crew we only need Director's name as rest crew members are again less known

So for this we had to fetch director's name from crew column

```
def fetch_director(obj):
  l=[]
  for i in ast.literal_eval(obj):
    if i['job'] == 'Director':
      l.append(i['name'])
  return l
```

Now , all the columns are in list , so we made overview column also into list

```
movies['overview']= movies['overview'].apply(lambda x : x.split())
```

Now there is space between first name and last name , so we have to remove them also. AS is it can create confusion

Like Science Fiction will convert into ScienceFiction , Sam Worthington into SamWorthington .

So , for this we used the following code :

```
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])
```

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

Now , we made our tag column , which we will use to find similarities between movies , which include a combination of all columns except movie_id and title .

movies['tag'] = movies['overview'] + movies['genres'] + movies['keywords'] + movies['cast'] + movies['crew']

Now we made new data_base where there are only 3 columns 'movie_id' , 'title' , 'tag'

new_df = movies[['movie_id' , 'title' , 'tag']]

Tag column is in list , so converting to string

new_df['tag'] = new_df['tag'].apply(lambda x : " ".join(x))

Now , make every word of the tag into lower case .

Then we apply stemming on each string as we want every word to be in their root form ,

Stemming is a natural language processing technique that reduces words to their base or root form, which can be useful for text processing and analysis. The NLTK library provides various stemmers, and in this case, you are using the PorterStemmer.

like actions , actions to be treated as action only , "running", "jumps", and "hoped", the result after applying stemming would be "run", "jump", and "hope", respectively.

Then we applied bag-of-words representation for the 'tag' column in the DataFrame
We are converting every movie tag to vectors using word of bag technique , also in vector words we haven't included stop words of english .

In this word of bag technique we first select most frequent say 5000 words and from each movie tag find how many time each movie tag word come

This will create a table of no.of movie * no.of frequent words size matrix , where each row is a vector of each movie

The code for the same is :

```
from sklearn.feature_extraction.text import CountVectorizer

cv = CountVectorizer(max_features=5000 , stop_words = 'english')
vector= cv.fit_transform(new_df['tag']).toarray()
```

max_features=5000: This sets the maximum number of features (words) to be extracted from the text. In this case, it will extract the 5000 most frequent words as features.

stop_words='english': This parameter specifies that common English stop words (e.g., "the", "and", "is", etc.) should be excluded from the features. These words are often considered to be less informative for many text analysis tasks.

vector :

```
array([[0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       ...,
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0]])
```

sparse matrix

Then we find cosine similarity matrix for each pair of movies

```
from sklearn.metrics.pairwise import cosine_similarity
similarity = cosine_similarity(vector)
```

Then used the pickle module in Python to save the DataFrame new_df and the cosine similarity matrix similarity as pickled files. Pickling is a way to serialize Python objects into a binary format that can be later deserialized to reconstruct the original objects.

Then made a main.py file , where I write code to make a streamlit website for the same

Key Components:

fetch_poster(movie_id): This function retrieves the movie poster image URL for a given movie ID using The Movie Database (TMDb) API. It takes the movie ID as input, makes an API request, and returns the full URL of the movie poster.

recommend(movie): This function takes the name of a movie as input and recommends five similar movies based on the cosine similarity matrix. It calculates the similarity scores between the input movie and all other movies in the dataset, sorts the movies based on similarity, and returns the names and poster URLs of the top five similar movies.

Loading Data: The script loads two pickled files using pickle.load() - 'movie_dict.pkl', which contains the movie data in a dictionary format, and 'similarity.pkl', which contains the cosine similarity matrix.

Streamlit App: The main part of the script builds the Streamlit app interface. It displays a title 'Movie Recommender System' and a dropdown menu to select a movie from the loaded dataset.

Recommendations: When the 'Recommend' button is clicked, the recommend() function is called with the selected movie's name as input. The function retrieves the recommended movie names and poster URLs and displays them in a grid using Streamlit's st.columns() and st.image() functions.

recommend(movie) : Function and is responsible for finding the indices of the top five most similar movies to the given input movie based on the cosine similarity scores. Let's break down the code:

Code :

```
def recommend(movie):
    movie_index = movies[movies['title'] == movie].index[0]
    distances = similarity[movie_index]
    movies_list = sorted(list(enumerate(distances)), reverse=True, key=lambda x: x[1])[1:6]
    recommended_list = []
    recommended_posters= []
    for i in movies_list:
        movie_id = movies.iloc[i[0]].movie_id

        recommended_list.append(movies.iloc[i[0]].title)
        # fetch poster from API
        recommended_posters.append(fetch_poster(movie_id))
    return recommended_list, recommended_posters
```

movie_index = movies[movies['title'] == movie].index[0]:
This line finds the index of the given input movie in the DataFrame movies. It first filters the rows of the DataFrame where the 'title' column matches the input movie name. Since there may be multiple rows with the same movie title, .index[0] is used to get the index of the first occurrence of the movie in the DataFrame. The variable movie_index stores this index, which will be used to retrieve the similarity scores for the input movie.

distances = similarity[movie_index]:
This line retrieves the cosine similarity scores between the input movie and all other movies in the similarity matrix similarity. The similarity matrix contains rows corresponding to different movies, and movie_index is used to access the row representing the input movie. The variable distances now store the array of similarity scores between the input movie and all other movies.

movies_list = sorted(list(enumerate(distances)), reverse=True, key=lambda x: x[1])[1:6]:
This line sorts the similarity scores in descending order and keeps the top five most similar movies (excluding the input movie itself). Let's break it down step-by-step:

enumerate(distances): This function pairs each similarity score with its corresponding index. The resulting pairs are then used for sorting.

sorted(..., reverse=True, key=lambda x: x[1]): The sorted() function is used to sort the similarity pairs in descending order based on the similarity scores (x[1]). This means that the movies with higher similarity scores will come first.

[1:6]: The code slices the sorted list to keep only the top five most similar movies (excluding the input movie itself). Index 0 in the sorted list would be the input movie, which is why it starts from index 1.

THANK YOU