MOVIE RECOMMENDATION SYSTEM

Project Work

Madhav Choudhary

- LinkedIn - Github

PROJECT FILES

DEMO VIDEO

MOVIE RECOMMENDED SYSTEM

1 . Introduction

The Movie Recommendation System is an AI-powered project designed to provide users with personalized movie suggestions based on their preferences. This project leverages natural language processing (NLP) and machine learning techniques to analyze the similarities between movies, allowing users to discover new films similar to ones they already enjoy. Built using Python, the system utilizes a content-based filtering approach that compares movies by extracting essential features like genres, keywords, cast, and crew. By transforming these textual attributes into vector representations using TF-IDF (Term Frequency-Inverse Document Frequency), the system calculates cosine similarity between movies, determining how closely related two movies are based on their descriptions.

The project is implemented in two main parts. First, in a Jupyter Notebook (movie recommender system.ipynb), where data preprocessing, feature engineering, and model training take place. After loading and cleaning data from publicly available movie datasets, the notebook generates similarity scores for each movie and saves them in a structured format for efficient access. The second component is an interactive web application (app.py) developed with Streamlit, which enables users to select a movie from a dropdown menu and instantly receive recommendations with accompanying posters retrieved from The Movie Database (TMDb) API. This interactive interface makes the recommendation process simple and engaging for the user. Overall, the project is a robust demonstration of how machine learning can enhance user experience in entertainment by making movie discovery more accessible, intuitive, and personalized.

2. Features

- **Content-Based Filtering**:
  Recommends movies based on the content (genre, cast, etc.) of the selected movie.

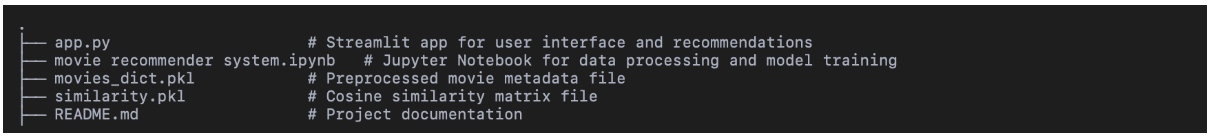
- **Interactive Web Application**:

Simple and intuitive interface powered by Streamlit.

- **Visual Enhancements**:

  Displays movie posters using The Movie Database (TMDb) API for a rich user experience.

3. Project Structure



```
.
├── app.py                     # Streamlit app for user interface and recommendations
├── movie recommender system.ipynb   # Jupyter Notebook for data processing and model training
├── movies_dict.pkl            # Preprocessed movie metadata file
├── similarity.pkl             # Cosine similarity matrix file
├── README.md                  # Project documentation
```

4. Installation and Usage

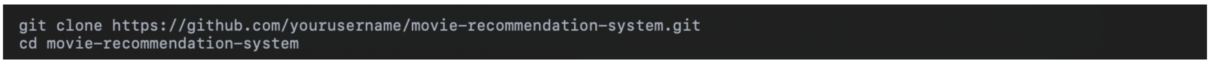
4.1. Prerequisites

Ensure you have the following installed:

- Python 3.8 or higher
- pip (Python package manager)

4.2. Setup Instructions

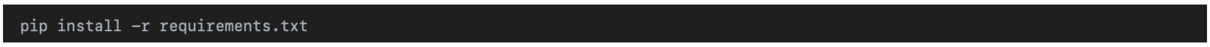
4.2.1 Clone the Repository:



```
git clone https://github.com/yourusername/movie-recommendation-system.git
cd movie-recommendation-system
```

4.2.2 Install Dependencies:

Install the necessary Python libraries using:



```
pip install -r requirements.txt
```

Note: The requirements.txt file should include the following libraries:

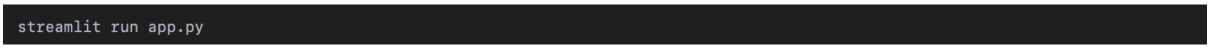
- streamlit
- pandas
- scikit-learn

- requests

4.2.3. Obtain TMDb API Key:

- Sign up on TMDb and obtain an API key.
- Add your API key to the app.py file in the fetch_poster function.

4.2.4. Run the Streamlit Application:



```
streamlit run app.py
```

4.2.5. Using the Application:

- Open your browser and navigate to the local URL provided by Streamlit.
- Select a movie from the dropdown menu to get personalized movie recommendations.

5. How It Works

5.1. Data Processing (movie recommender system.ipynb):

- Loads and preprocesses movie data.
- Generates a “tags” column for each movie, combining its genre, cast, and other details.
- Converts this text data into vectors using TF-IDF (Term Frequency-Inverse Document Frequency) to facilitate similarity measurement.
- Computes cosine similarity between movies and saves the processed data and similarity matrix.

5.2. Web Application (app.py):

- Loads precomputed data and similarity matrix.

- Provides an interface where users can choose a movie and receive similar movie suggestions.
- Fetches movie posters using the TMDb API and displays them alongside recommendations.

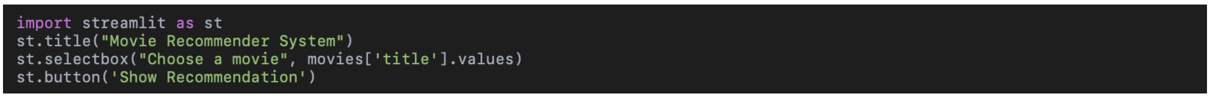
6. Future Enhancements

- **Hybrid Recommendation**: Combine content-based filtering with collaborative filtering for improved recommendations.
- **User Profiles**: Store user preferences and create personalized recommendation lists.
- **Improved Similarity Calculation**: Experiment with other NLP techniques, such as Word2Vec, for vectorization.

7. Frameworks and Libraries Used :

7.1. Streamlit

- **Definition**: Streamlit is an open-source Python library that allows for the quick creation of interactive, web-based applications for data science and machine learning models.
- **Usage in** app.py:
  - Used to build a user-friendly interface for the movie recommendation system, including widgets like dropdowns, buttons, and displaying text and images.
  - Allows users to select a movie and view recommended movies with their posters in a streamlined, interactive manner.



```
import streamlit as st
st.title("Movie Recommender System")
st.selectbox("Choose a movie", movies['title'].values)
st.button('Show Recommendation')
```

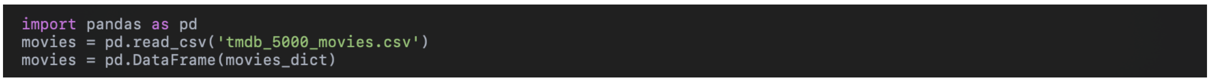
7.2. Pandas

- **Definition**: Pandas is a popular data manipulation and analysis library in Python. It provides data structures and functions to work with structured data efficiently.

- **Usage in Both Files**:

  - Used to load and manipulate the movie and credits datasets.

  - In movie recommender system.ipynb, it helps merge datasets and preprocess data into a usable format.
  - In app.py, it's used to load the preprocessed movie data from a serialized pickle file into a DataFrame for convenient data handling.



```
import pandas as pd
movies = pd.read_csv('tmdb_5000_movies.csv')
movies = pd.DataFrame(movies_dict)
```

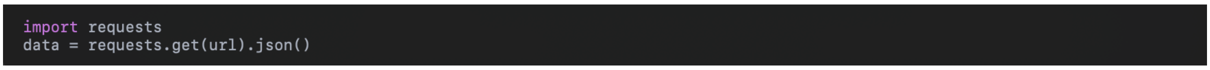
7.3. Requests

- **Definition**: Requests is a Python library used for making HTTP requests to access data from APIs.

- **Usage in** app.py:

  - Used to send GET requests to the TMDb (The Movie Database) API to fetch poster images for recommended movies.

  - Retrieves JSON data containing the poster path and other details of the selected movies from the TMDb API.



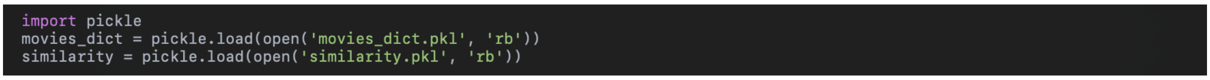
```
import requests
data = requests.get(url).json()
```

7.4. Pickle

- **Definition**: Pickle is a Python module used to serialize (save) and deserialize (load) Python objects, making it useful for storing precomputed data that doesn't need to be recalculated every time.

- **Usage in Both Files**:

  - In movie recommender system.ipynb, pickle is used to save the preprocessed movie data and similarity matrix.
  - In app.py, these pickled files are loaded to access the data without recalculating it , improving the efficiency of the Streamlit app.



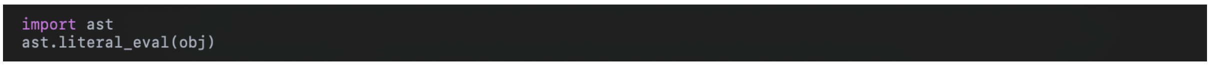
```
import pickle
movies_dict = pickle.load(open('movies_dict.pkl', 'rb'))
similarity = pickle.load(open('similarity.pkl', 'rb'))
```

7.5. AST (Abstract Syntax Trees)

- **Definition**: AST is a Python module that helps parse Python expressions, such as those in JSON-like structures, into usable data.

- **Usage in** movie recommender system.ipynb:

  - The literal_eval function from AST is used to safely evaluate strings containing lists or dictionaries, such as JSON-formatted columns in the dataset.
  - This makes it easier to work with complex, nested structures by converting them into lists or dictionaries.



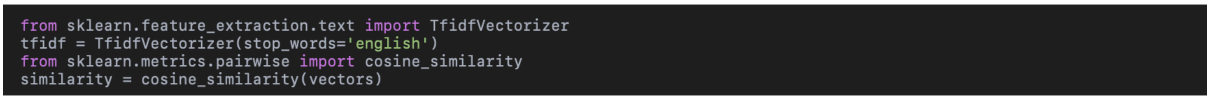
```
import ast
ast.literal_eval(obj)
```

7.6. Scikit-Learn (sklearn)

- **Definition**: Scikit-Learn is a machine learning library in Python, widely used for data preprocessing, model building, and evaluation. It provides efficient tools for machine learning and statistical modeling.

- **Usage in** movie recommender system.ipynb:

  - **TfidfVectorizer**: This module from sklearn.feature_extraction.text is used to convert text into numerical vectors by calculating the TF-IDF (Term Frequency-Inverse Document Frequency) values for each term in the dataset. It's essential for processing movie tags to determine how similar they are.
  - **cosine_similarity**: This function from sklearn.metrics.pairwise computes the cosine similarity between vectors. In this project, it's used to measure similarity between movie vectors generated from the TF-IDF transformation, which is then used to find and rank similar movies.



```
from sklearn.feature_extraction.text import TfidfVectorizer
tfidf = TfidfVectorizer(stop_words='english')
from sklearn.metrics.pairwise import cosine_similarity
similarity = cosine_similarity(vectors)
```

7.7. NumPy

- **Definition**: NumPy is a powerful numerical computing library in Python that provides support for large, multi-dimensional arrays and matrices, along with a variety of mathematical functions to operate on these arrays.
- **Usage in** movie recommender system.ipynb:
  - NumPy is often used implicitly through Scikit-Learn functions (like TfidfVectorizer and cosine_similarity), which rely on NumPy arrays for efficient matrix operations.
  - This project specifically doesn't import NumPy directly but benefits from its optimizations through other libraries.

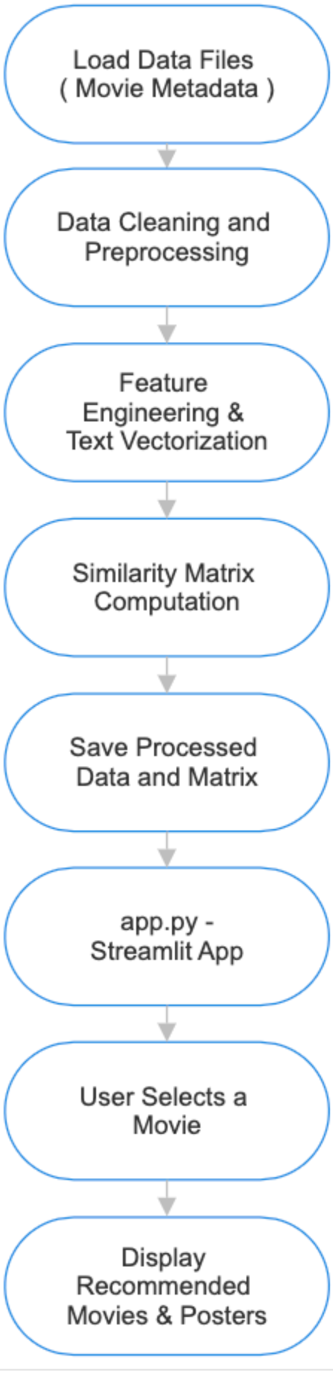
8. Summary of Frameworks and Libraries

Each library in this project serves a distinct purpose:

- **Streamlit**: Builds an interactive user interface in app.py.
- **Pandas**: Handles data loading, merging, and processing in both files.
- **Requests**: Fetches data from the TMDb API in app.py to display movie posters.
- **Pickle**: Serializes and deserializes data for efficient access and performance in both files.
- **AST**: Parses complex JSON-like strings into usable data formats in movie recommender system.ipynb.
- **Scikit-Learn**: Transforms movie tags to vectors using TF-IDF and computes cosine similarity for movie recommendations.
- **NumPy**: Used implicitly for efficient array handling and mathematical operations, particularly with Scikit-Learn.

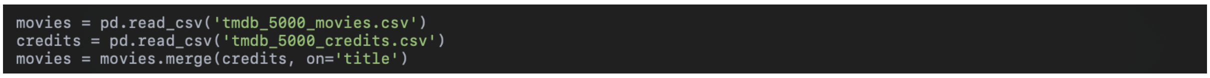
These frameworks and libraries work together to create a seamless, data-driven movie recommendation system, from preprocessing data and computing similarities in the Jupyter Notebook to providing an interactive user experience in the Streamlit application.

9. Steps Followed :-


Load Data Files
(Movie Metadata)
Data Cleaning and Preprocessing
Feature Engineering & Text Vectorization
Similarity Matrix Computation
Save Processed Data and Matrix
app.py - Streamlit App
User Selects a Movie
Display Recommended Movies & Posters

9.1. Project Setup and Data Loading

- **Definition**: The data loading stage involves reading datasets containing movie information. The movie and credits data are merged, providing a single source for all essential movie details.

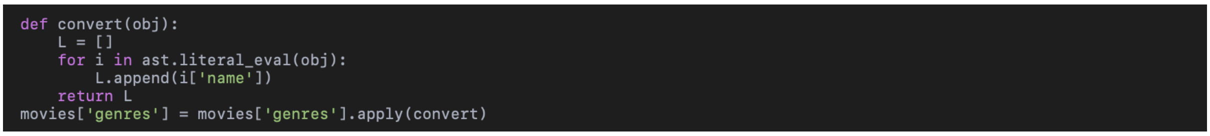


```
movies = pd.read_csv('tmdb_5000_movies.csv')
credits = pd.read_csv('tmdb_5000_credits.csv')
movies = movies.merge(credits, on='title')
```

- **Explanation**: Here, pd.read_csv loads the movie data from CSV files, while merge joins the two datasets based on the common “title” column. This results in a consolidated dataset with both movie information and crew details.

9.2. Data Preprocessing

- **Definition**: Preprocessing involves transforming data into a format suitable for analysis. In this system, specific features like genres, cast, and crew are extracted and stored in a new “tags” column.

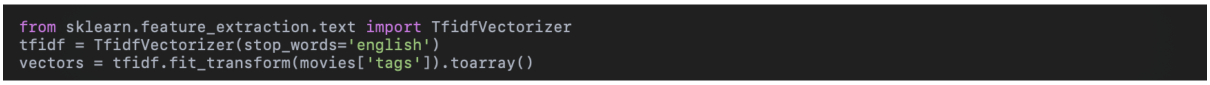


```
def convert(obj):
    L = []
    for i in ast.literal_eval(obj):
        L.append(i['name'])
    return L
movies['genres'] = movies['genres'].apply(convert)
```

- **Explanation**: The convert function iterates through JSON-like strings in columns (e.g., genres, keywords). Using ast.literal_eval, it safely parses each JSON string, extracting only relevant names. This structure improves similarity calculations.

9.3. Text Vectorization with TF-IDF

- **Definition**: Text vectorization converts text data into numerical form, which is crucial for calculating similarities between movies. TF-IDF (Term Frequency-Inverse Document Frequency) is used to give more weight to important terms within movie “tags.”

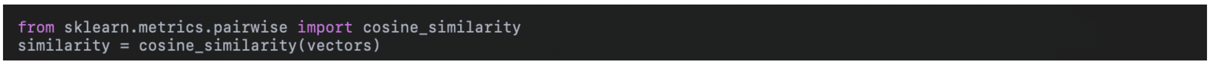


```python
from sklearn.feature_extraction.text import TfidfVectorizer
tfidf = TfidfVectorizer(stop_words='english')
vectors = tfidf.fit_transform(movies['tags']).toarray()
```

- **Explanation**: The TfidfVectorizer removes common words (stop words) and assigns weights to terms. fit_transform converts each movie's tags into a TF-IDF vector, and toarray() finalizes this as a matrix for similarity calculations.

9.4. Similarity Matrix Calculation

- **Definition**: A similarity matrix helps identify movies with similar tags by measuring the cosine similarity between movie vectors. Cosine similarity calculates how close two vectors are by measuring the cosine of the angle between them.

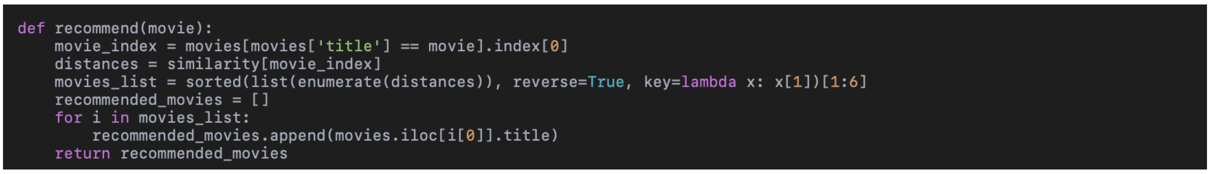


```python
from sklearn.metrics.pairwise import cosine_similarity
similarity = cosine_similarity(vectors)
```

- **Explanation**: Using cosine_similarity, the vectors are compared, and a similarity score is assigned between pairs of movies. This similarity matrix (stored in similarity) is later used to find the top recommendations.

9.5. Movie Recommendation Function

- **Definition**: The recommendation function uses the similarity matrix to identify and return movies similar to a user-selected movie.

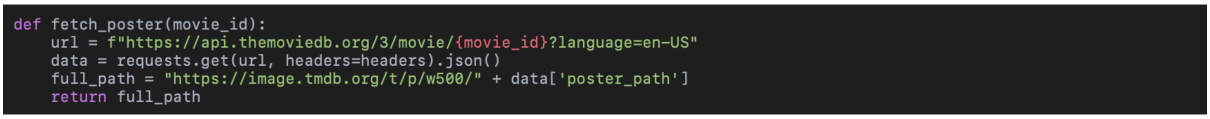


```python
def recommend(movie):
    movie_index = movies[movies['title'] == movie].index[0]
    distances = similarity[movie_index]
    movies_list = sorted(list(enumerate(distances)), reverse=True, key=lambda x: x[1])[1:6]
    recommended_movies = []
    for i in movies_list:
        recommended_movies.append(movies.iloc[i[0]].title)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

- **Explanation**: In this function:
    - movie_index finds the index of the selected movie.
    - distances retrieves similarity scores for this movie.
    - movies_list sorts the scores in descending order, picking the top 5 movies with the highest similarity.
    - Finally, the titles of these movies are collected in recommended_movies.

9.6. Fetching Movie Posters

- **Definition**: To enhance user experience, the system fetches posters for recommended movies from an external API (TMDb).

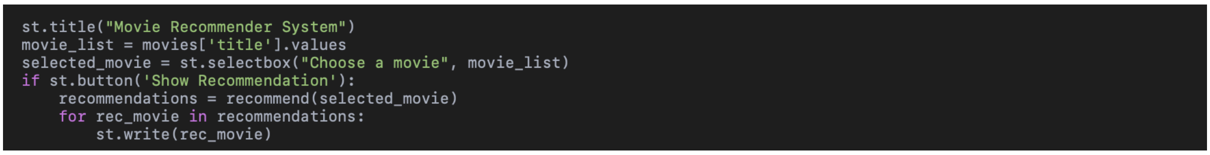


```
def fetch_poster(movie_id):
    url = f"https://api.themoviedb.org/3/movie/{movie_id}?language=en-US"
    data = requests.get(url, headers=headers).json()
    full_path = "https://image.tmdb.org/t/p/w500/" + data['poster_path']
    return full_path
```

- **Explanation**: Here, requests.get makes a GET request to TMDb, using movie_id to retrieve the poster path. The poster's full URL is constructed using a base URL plus the poster path, allowing the app to display relevant images with recommendations.

9.7. Building the User Interface with Streamlit

- **Definition**: The UI enables users to interact with the system, select a movie, and view recommendations along with their posters.

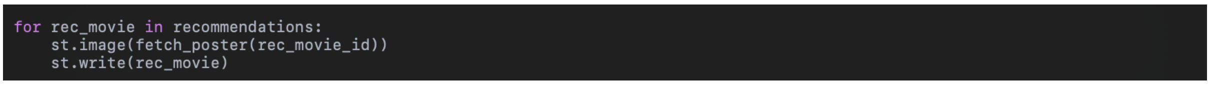


```
st.title("Movie Recommender System")
movie_list = movies['title'].values
selected_movie = st.selectbox("Choose a movie", movie_list)
if st.button('Show Recommendation'):
    recommendations = recommend(selected_movie)
    for rec_movie in recommendations:
        st.write(rec_movie)
```

- **Explanation**:
  - st.title sets the app title.
  - st.selectbox provides a dropdown for movie selection.
  - When the "Show Recommendation" button is clicked, recommend generates recommendations based on the selected movie.
  - Each recommendation is displayed via st.write.

9.8. Displaying Movie Posters

- **Definition**: Using Streamlit's image display functionality, posters are shown next to each recommended movie, enhancing the app's visual appeal.



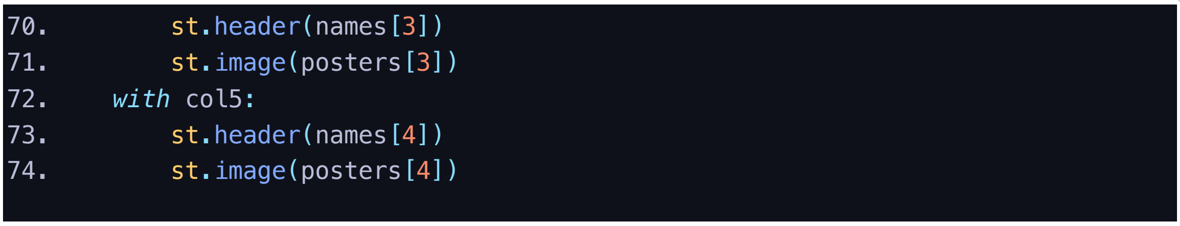
```
for rec_movie in recommendations:
    st.image(fetch_poster(rec_movie_id))
    st.write(rec_movie)
```

- **Explanation**: For each recommended movie, fetch_poster retrieves the image, and st.image displays it. This visual representation aids users in identifying movies by cover images.

10. CODE FILE - 1

10.1 app.py - Streamlit Application for Movie Recommendations

```python
import streamlit as st
import pickle
import pandas as pd
import requests

headers = {
    "accept": "application/json",
    "Authorization": "Bearer eyJhbGciOiJIUzI1NiJ9.eyJhdWQiOiI4Njg5M2Y4ZDI5MDQ4MzAzZTEwMmYyZDJmNzFjODljYyIsInN1YiI6IjY0YTk0ZThjYjY4NmI5MDBlZGY4ZTI0OCIsInNjb3BlcyI6WyJhcGlfcmVhZCJdLCJ2ZXJzaW9uIjoxfQ.hyyMVyPgfuEsCrP5SE6VrjLP4hR53SKxA2b28Ok9goU"
}
def fetch_poster(movie_id):
    url = "https://api.themoviedb.org/3/movie/{}?language=en-US".format(movie_id)
    data = requests.get(url, headers=headers)
    data = data.json()
    poster_path = data['poster_path']
    full_path = "https://image.tmdb.org/t/p/w500/" + poster_path
    # full_path = "https://image.tmdb.org/t/p/w500/1E5baAaEse26fej7uHcjOgEE2t2.jpg"
    return full_path

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_movies = []
    # recommended_movies_posters = []
    for i in movies_list:
        movie_id = movies.iloc[i[0]].movie_id
        recommended_movies.append(movies.iloc[i[0]].title)
        # fetch poster from API
        # recommended_movies_posters.append(fetch_poster(movie_id))
    return recommended_movies

movies_dict = pickle.load(open('movies_dict.pkl','rb'))
movies = pd.DataFrame(movies_dict)

similarity = pickle.load(open('similarity.pkl','rb'))

st.set_page_config(layout='wide')

st.title('Movie Recommender System')

selected_movie_name = st.selectbox(
    'Type or select a movie from the dropdown?',
    movies['title'].values
)
if st.button('Recommend'):
    names = recommend(selected_movie_name)
    posters = [
        "https://t3.ftcdn.net/jpg/04/38/03/50/360_F_438035062_bxhey1N5fbRvjgPY0O7SqOnq3VzlrrSJ.jpg",
        "https://img.freepik.com/premium-photo/dark-blue-navy-color-scheme-photo-background-picture-day-background_136558-4027.jpg",
        "https://img.freepik.com/free-photo/green-gradient-abstract-background-empty-room-with-space-your-text-picture_1258-54428.jpg?size=626&ext=jpg&ga=GA1.1.1224184972.1714694400&semt=ais",
        "https://img.freepik.com/free-photo/orange-background_23-2147674307.jpg",
        "https://media.istockphoto.com/id/540533738/photo/blue-background.jpg?s=612x612&w=0&k=20&c=X3doabKNqbK6hBhq7vq5c8zQOYwycVFubOkxZC9NMoc="
    ]
    col1,col2,col3 = st.columns(3)

    with col1:
        st.header(names[0])
        st.image(posters[0])
    with col2:
        st.header(names[1])
        st.image(posters[1])
    with col3:
        st.header(names[2])
        st.image(posters[2])

    col4, col5 = st.columns(2)

    with col4:
```



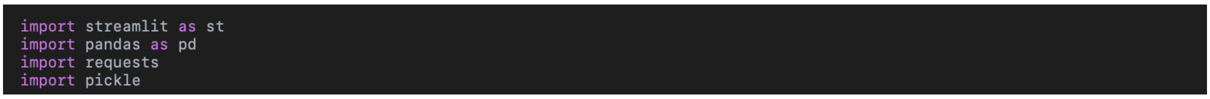
```
        st.header(names[3])
        st.image(posters[3])
    with col5:
        st.header(names[4])
        st.image(posters[4])
```

The app.py file is the main script for the Streamlit application. It builds an interactive interface where users can select a movie and receive recommendations with associated posters. This script leverages Streamlit's widgets and integrates the recommendation model to deliver results in a user-friendly format.

10.1.1 Key Sections and Code Explanation

10.1.1.1. Setting Up Imports and Loading Libraries

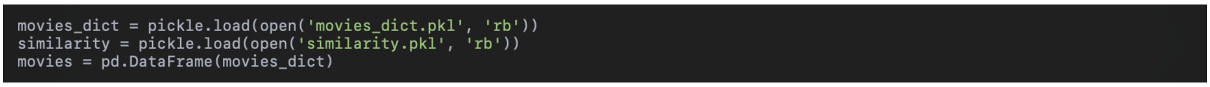
This part imports necessary libraries like Streamlit (st), Pandas, and additional utilities to load precomputed data and make API requests.



```
import streamlit as st
import pandas as pd
import requests
import pickle
```

10.1.1.2. Loading Precomputed Data and Model Files

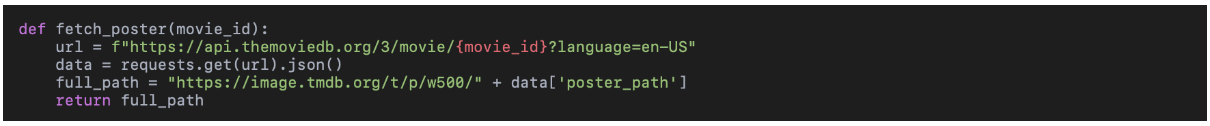
Here, the application loads the movies_dict and similarity objects (created and saved in the Jupyter Notebook). The pickle module is used to deserialize the data, enhancing performance by avoiding real-time model training.



```
movies_dict = pickle.load(open('movies_dict.pkl', 'rb'))
similarity = pickle.load(open('similarity.pkl', 'rb'))
movies = pd.DataFrame(movies_dict)
```

10.1.1.3. Fetching Movie Posters

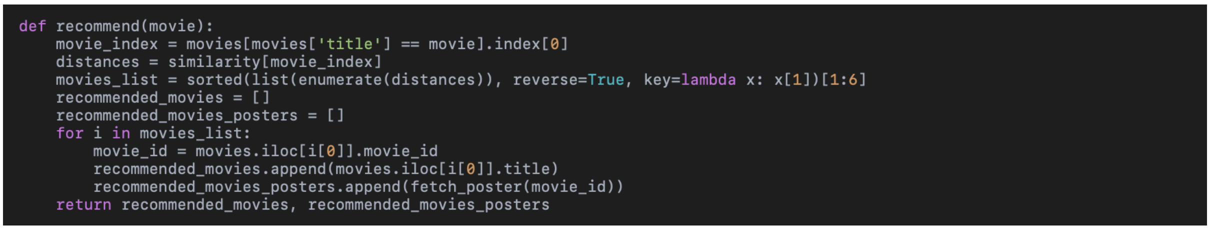
The fetch_poster function interacts with The Movie Database (TMDb) API to retrieve the poster image for a given movie. It makes a request using the movie_id, parses the JSON response, and constructs the full URL for the poster.



```
def fetch_poster(movie_id):
    url = f"https://api.themoviedb.org/3/movie/{movie_id}?language=en-US"
    data = requests.get(url).json()
    full_path = "https://image.tmdb.org/t/p/w500/" + data['poster_path']
    return full_path
```

10.1.1.4. Recommendation Function

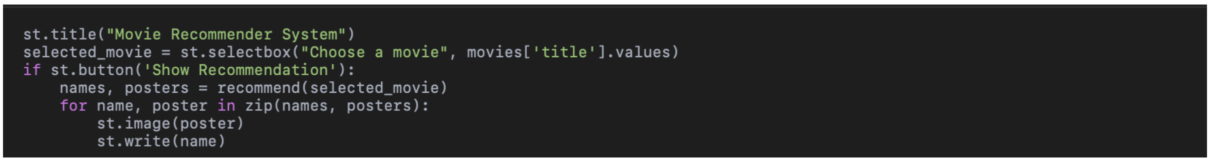
The recommend function identifies the most similar movies based on cosine similarity. For a selected movie, the function finds similar movies using precomputed similarity scores and returns titles and poster URLs for display.



```
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_movies = []
    recommended_movies_posters = []
    for i in movies_list:
        movie_id = movies.iloc[i[0]].movie_id
        recommended_movies.append(movies.iloc[i[0]].title)
        recommended_movies_posters.append(fetch_poster(movie_id))
    return recommended_movies, recommended_movies_posters
```

10.1.1.5. Streamlit Interface Setup

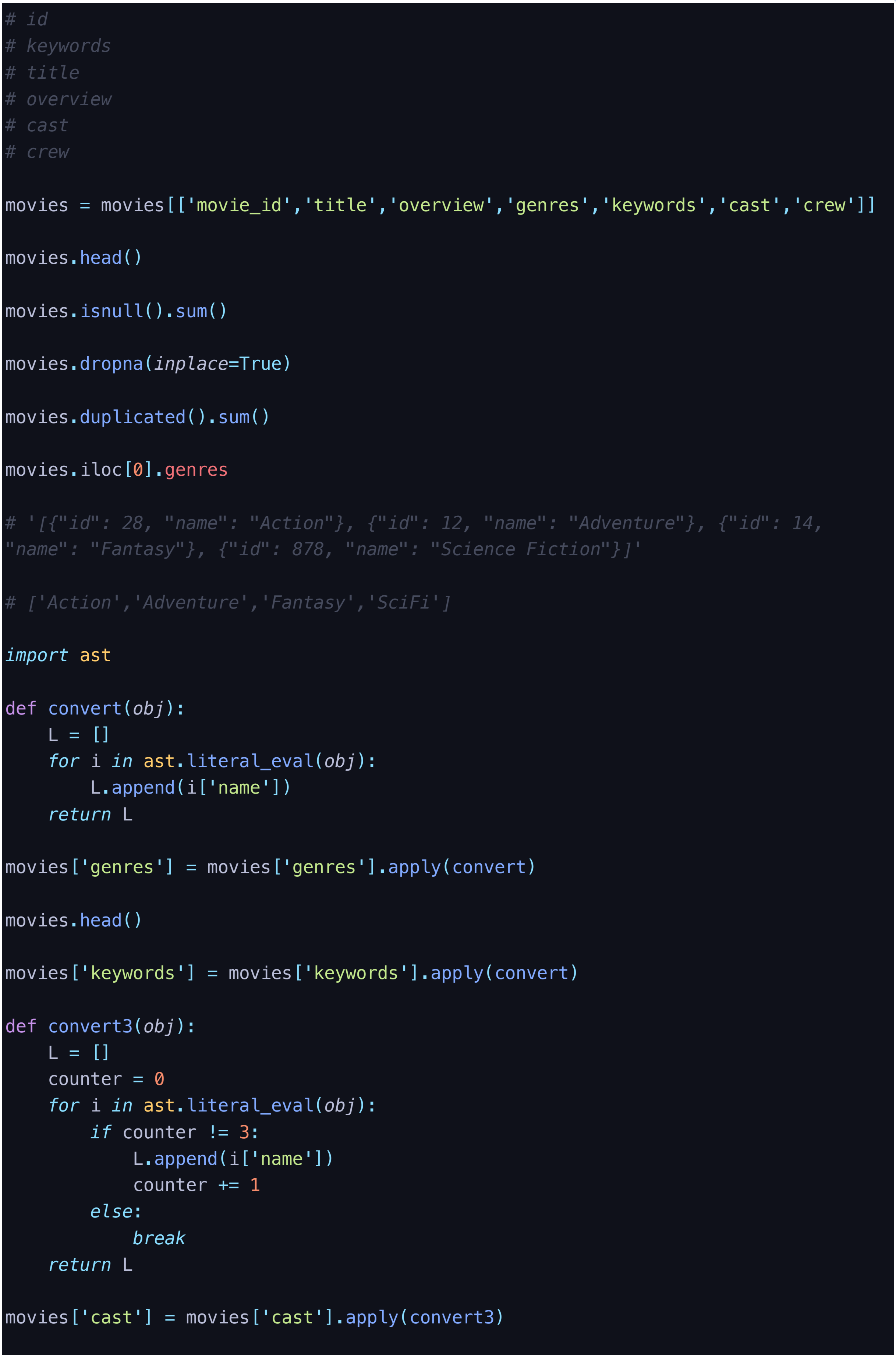
This section defines the application interface with Streamlit. It includes a title, dropdown menu for movie selection, and a recommendation display area. When the user clicks "Show Recommendation," the selected movie is passed to recommend(), and the recommended movies and posters are shown.



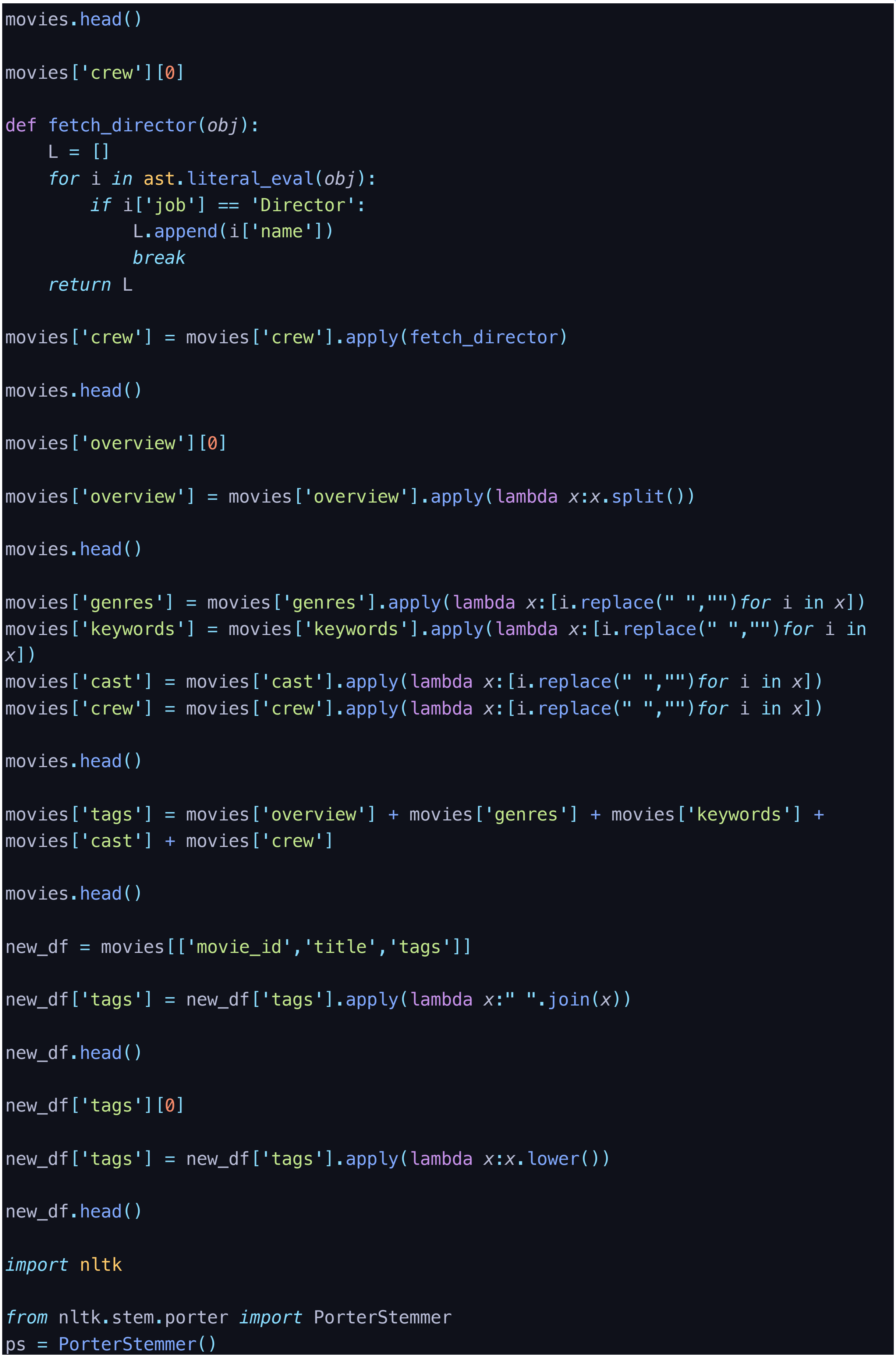
```
st.title("Movie Recommender System")
selected_movie = st.selectbox("Choose a movie", movies['title'].values)
if st.button('Show Recommendation'):
    names, posters = recommend(selected_movie)
    for name, poster in zip(names, posters):
        st.image(poster)
        st.write(name)
```

10.2. movie recommender system.ipynb - Data Preparation and Model Development

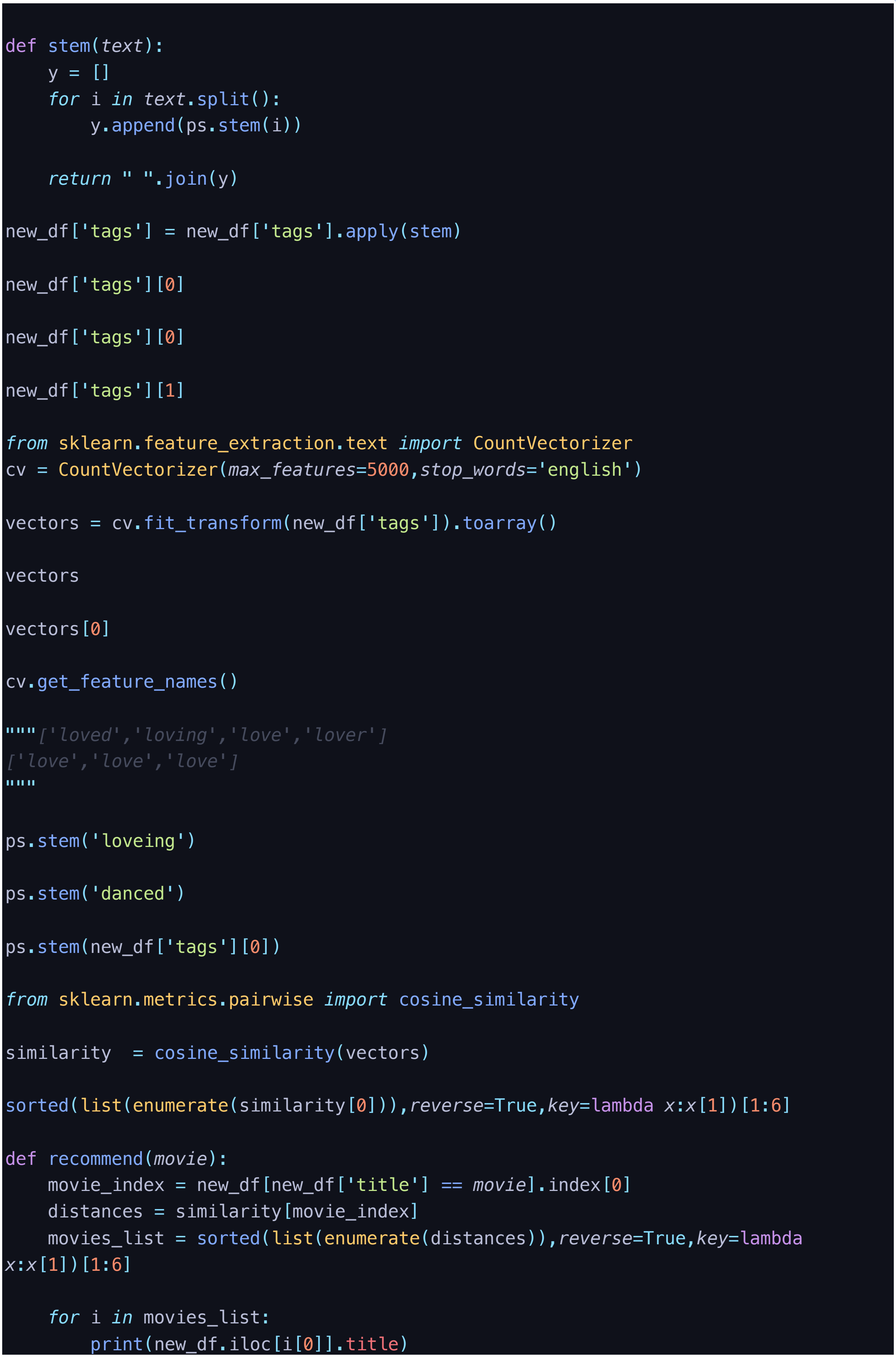
```
# -*- coding: utf-8 -*-
"""movie recommender system.ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1U5lyghla5FlQRqQXJBJ7IUNFuBhqOQBV
"""

import numpy as np
import pandas as pd

movies = pd.read_csv('tmdb_5000_movies.csv')
credits = pd.read_csv('tmdb_5000_credits.csv')

movies.head(1)

credits.head(1)

movies = movies.merge(credits,on='title')

movies.head(1)

# genres
```



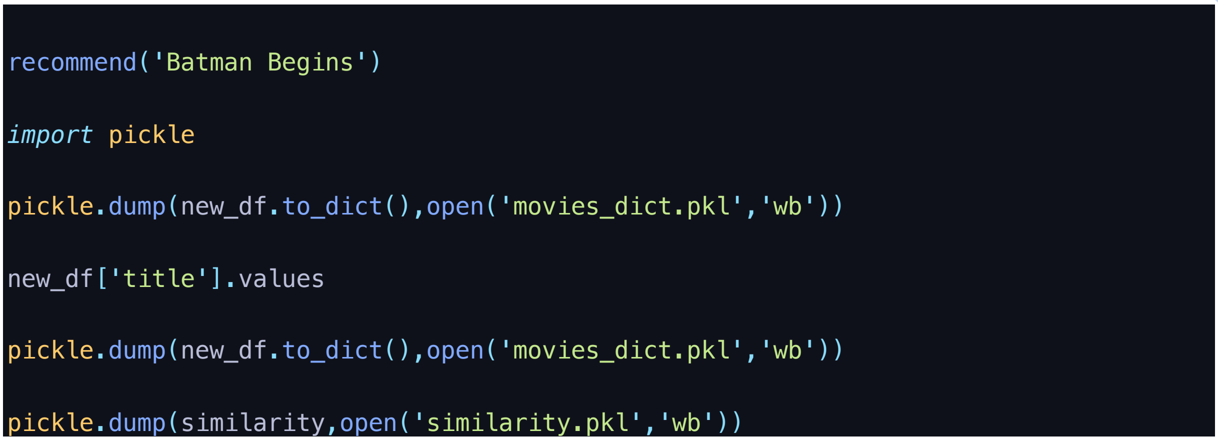
```
# id
# keywords
# title
# overview
# cast
# crew

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

movies.head()

movies.isnull().sum()

movies.dropna(inplace=True)

movies.duplicated().sum()

movies.iloc[0].genres

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

# ['Action','Adventure','Fantasy','SciFi']

import ast

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

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

movies.head()

movies['keywords'] = movies['keywords'].apply(convert)

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

movies['cast'] = movies['cast'].apply(convert3)
```



```
movies.head()

movies['crew'][0]

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

movies['crew'] = movies['crew'].apply(fetch_director)

movies.head()

movies['overview'][0]

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

movies.head()

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])
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.head()

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

movies.head()

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

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

new_df.head()

new_df['tags'][0]

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

new_df.head()

import nltk

from nltk.stem.porter import PorterStemmer
ps = PorterStemmer()
```



```
def stem(text):
    y = []
    for i in text.split():
        y.append(ps.stem(i))

    return " ".join(y)

new_df['tags'] = new_df['tags'].apply(stem)

new_df['tags'][0]

new_df['tags'][0]

new_df['tags'][1]

from sklearn.feature_extraction.text import CountVectorizer
cv = CountVectorizer(max_features=5000,stop_words='english')

vectors = cv.fit_transform(new_df['tags']).toarray()

vectors

vectors[0]

cv.get_feature_names()

"""['loved','loving','love','lover']
['love','love','love']
"""

ps.stem('loveing')

ps.stem('danced')

ps.stem(new_df['tags'][0])

from sklearn.metrics.pairwise import cosine_similarity

similarity  = cosine_similarity(vectors)

sorted(list(enumerate(similarity[0])),reverse=True,key=lambda x:x[1])[1:6]

def recommend(movie):
    movie_index = new_df[new_df['title'] == movie].index[0]
    distances = similarity[movie_index]
    movies_list = sorted(list(enumerate(distances)),reverse=True,key=lambda
x:x[1])[1:6]

    for i in movies_list:
        print(new_df.iloc[i[0]].title)
```



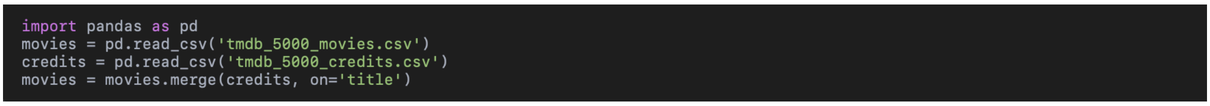
```
recommend('Batman Begins')

import pickle

pickle.dump(new_df.to_dict(),open('movies_dict.pkl','wb'))

new_df['title'].values

pickle.dump(new_df.to_dict(),open('movies_dict.pkl','wb'))

pickle.dump(similarity,open('similarity.pkl','wb'))
```

The movie recommender system.ipynb notebook is where data preprocessing, feature engineering, and model creation occur. This script sets up the recommendation engine, calculates movie similarities, and saves precomputed data for fast retrieval in the app.py file.

10.2.1. Key Sections and Code Explanation

10.2.1.1. Data Loading and Initial Exploration

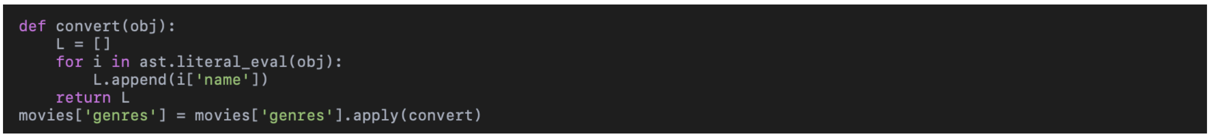
The notebook begins by loading the tmdb_5000_movies.csv and tmdb_5000_credits.csv files using Pandas. It then merges these datasets on the “title” column, producing a dataset with comprehensive movie details, including crew, genres, and keywords.



```
import pandas as pd
movies = pd.read_csv('tmdb_5000_movies.csv')
credits = pd.read_csv('tmdb_5000_credits.csv')
movies = movies.merge(credits, on='title')
```

10.2.1.2. Data Preprocessing

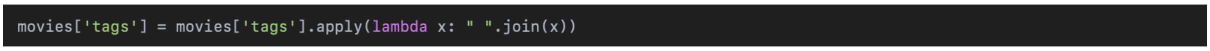
Several columns, including genres, keywords, cast, and crew, contain JSON-like strings. Custom functions are used to parse these strings and extract relevant information (e.g., actor names, genres). These details are combined into a single “tags” column to represent each movie’s content.



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

10.2.1.3. Feature Engineering

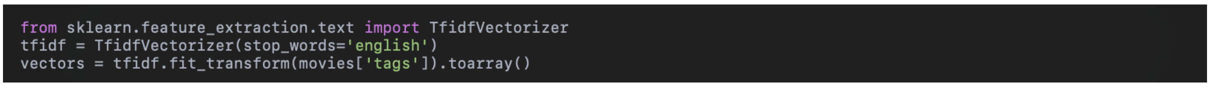
After extracting keywords, cast, and crew names, each movie's "tags" column is cleaned by removing spaces, duplicates, and transforming the text into lowercase. This text processing ensures that similar terms appear consistently in the vector representation.



```
movies['tags'] = movies['tags'].apply(lambda x: " ".join(x))
```

10.2.1.4. Text Vectorization with TF-IDF

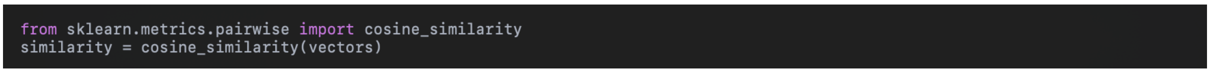
TF-IDF vectorization converts the "tags" column into vectors. The TfidfVectorizer assigns higher weights to unique words and lower weights to frequent ones, enabling the model to focus on distinguishing features of each movie.



```
from sklearn.feature_extraction.text import TfidfVectorizer
tfidf = TfidfVectorizer(stop_words='english')
vectors = tfidf.fit_transform(movies['tags']).toarray()
```

10.2.1.5. Computing Cosine Similarity

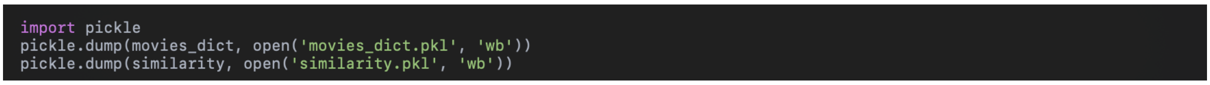
The cosine similarity matrix is computed using the TF-IDF vectors. This matrix indicates the similarity between each pair of movies based on their tags, providing the basis for content-based recommendations.



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

10.2.1.6.Saving Processed Data

The processed data, including the movies_dict and similarity matrix, is saved as pickle files. These files are loaded in app.py to avoid recomputation and make the app responsive.



```
import pickle
pickle.dump(movies_dict, open('movies_dict.pkl', 'wb'))
pickle.dump(similarity, open('similarity.pkl', 'wb'))
```

10.3. Summary

10.3.1. Summary of app.py

The app.py file is a Streamlit-based application that provides a graphical interface for the movie recommendation system. It loads precomputed similarity data and movie metadata, allowing users to select a movie and see similar movies with poster images in an interactive and visually engaging way.

10.3.2. Summary of movie recommender system.ipynb

The notebook movie recommender system.ipynb prepares data, processes text-based features, and builds the similarity matrix essential for the recommendation system. This file completes model training, saves preprocessed data, and provides the core similarity data used by the Streamlit application.

10.3.3. Overall Summary

Together, app.py and movie recommender system.ipynb create an AI-driven Movie Recommendation System. The Jupyter Notebook handles the data preparation and model training stages, producing precomputed data files used by app.py. The app.py script then provides an interactive interface where users select a movie and receive recommendations with visuals. This design enables efficient and accurate movie suggestions in a responsive, web-based application.

11. CONCLUSION :

The Movie Recommendation System exemplifies how data science and machine learning can transform media consumption by making recommendations more precise and user-centric. Through this project, a comprehensive content-based recommendation engine was developed, which leverages text processing, vectorization, and similarity calculations to identify movies related by theme, genre, or cast. The use of frameworks like Pandas, Scikit-Learn, and Streamlit not only allowed efficient data processing but also facilitated a seamless deployment of the model in an intuitive, web-based application. By integrating the TMDb API, the app enhances user engagement by providing visually appealing movie posters, making the recommendation process both informative and aesthetically pleasing.

This system can serve as a foundation for more sophisticated recommendation engines. Potential extensions could involve integrating collaborative filtering techniques or hybrid models that combine multiple data sources to further refine recommendation quality. Additionally, the streamlined, modular code and interface allow for easy expansion, such as including user rating data or exploring real-time user interaction data. As a learning experience, the project also illustrates the end-to-end workflow of machine learning projects, from data preprocessing to model deployment, and highlights the importance of both accuracy and user experience in delivering effective AI solutions. This project ultimately underscores the transformative power of AI in creating personalized experiences in entertainment and beyond.