CONTENT BASED MOVIE RECOMMENDATION SYSTEM

PROJECT BRIEFING: This project focuses on building a "Content-Based" Recommender System. This system recommends movies by analyzing the metadata of the movies themselves such as **genres**, **keywords**, **cast**, and **crew**. The goal is to provide a seamless user experience where choosing one movie leads to 5 highly relevant suggestions based on similar thematic content using **cosine similarity.**

TECH STACK: Python, Pandas, Scikit-Learn, NLTK, Streamlit

Visit: https://93fyn9be4jgcaefouqefs8.streamlit.app/

Dataset: TMDB 5000 Movie Dataset (Kaggle)

- `tmdb_5000_movies.csv`: Contains budget, genres, overview, and popularity.
- `tmdb_5000_credits.csv`: Contains movie IDs, titles, cast (actors), and crew (directors/producers).

The first step involved merging the two datasets by **'title'** column. This resulted in a combined dataframe containing all necessary attributes for content analysis.

Feature Selection: Out of 23 available columns, the following were selected for their importance:

1. **Movie_id:** For fetching posters in the UI.
2. **Title:** The primary identifier.
3. **Overview, Genres, Keywords, Cast, Crew:** The metadata used to build the recommendation engine.

To make the data readable by a machine learning model, overview, genre, keywords, cast, crew columns were transformed into a single unified column called **`tags`**.

Data Cleaning Steps:

1. **JSON Parsing:** Used the **`ast.literal_eval`** function to convert string-formatted lists (like genres and keywords) into Python lists.

2. **Space Removal:** Convert names like "Nabil Adnan" into "NabilAdnan" This ensures the model treats the full name as a single unique entity rather than two separate words.

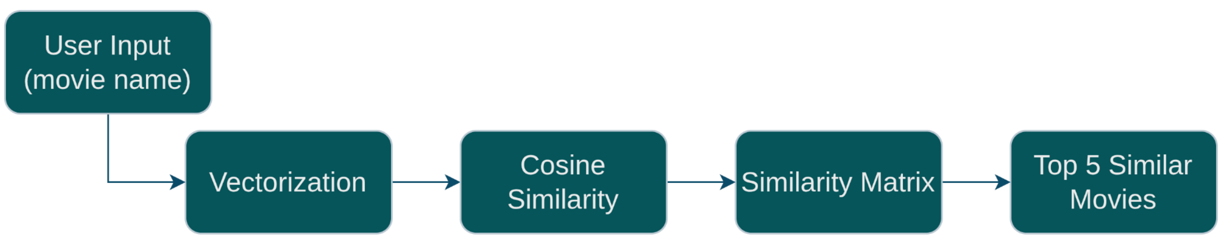
3. **Stemming:** Applied the **PorterStemmer** from the **NLTK** library. This reduces words to their root form (e.g., "loving," "loved," and "loves" all become "love"), which prevents the model from treating different forms of the same word as separate features.

Text to Numbers: I utilized **CountVectorizer** (Bag of Words) to convert the **`tags`** column into a 5000-dimensional vector space.

Cosine Similarity: Instead of using Euclidean distance (which measures the straight line between points), I used **Cosine Similarity**. This calculates the angle between two movie vectors. If the angle is small, the movies are highly similar.

Flow:

1. User inputs a movie name
2. System find the index of that movie
3. System looks into the similarity matrix
4. Sorts the highest similarity scores and returns the top 5 similar movies.



The Python function **`recommend(movie)`** handles the logic. It performs a reverse mapping of indexes to movie titles to display the final results to the user.

Exporting the Model: To make the project "production-ready" for a Streamlit web app, I used the **`pickle`** library to save the processed data and the similarity matrix as binary files:

1. **`movies.pkl`**: Stores the movie list and dictionary.
2. **`similarity.pkl`**: Stores the pre-calculated 5000-dimensional similarity scores for instant retrieval.

User Interface (Streamlit): The project was deployed as a web application where users can select a movie from a dropdown menu. The app then fetches the similarity scores and displays the top 5 recommended movies instantly.

Final Performance: The model successfully identifies thematic links. For example, searching for "The Dark Knight Rises" accurately recommends other Batman films and high-action superhero dramas, proving the effectiveness of the Content-Based filtering approach.