**Documentation of Movie Recommendation System**

**Basic Movie recommendation System**

Data Source:

* The dataset used is [Bollywood\_full.csv](https://www.kaggle.com/datasets/dell4010/bollywood-movies-19502019).

Libraries to Import:

* Numpy
* Pandas
* Matplotlib
* Seaborn

Importing the Dataset:

1. Import the dataset into a DataFrame named movies.
2. Perform the following checks:
   * Shape
   * Info

Columns in the movies DataFrame:

The dataset contains the following columns:

* title\_x, imdb\_id, poster\_path, wiki\_link, title\_y, original\_title, is\_adult, year\_of\_release, runtime, genres, imdb\_rating, imdb\_votes, story, summary, tagline, actors, wins\_nominations, release\_date.

Data Size:

* The dataset consists of 18 columns and 4329 rows.
* Movies listed range from the years 1950 to 2019.

Columns to Use for the Model:

We will select the following columns for the recommendation model:

* Title: We will use only one title column.
* imdb\_id: Unique identifier for the movie.
* is\_adult: Important for filtering movies based on family preferences.
* Year of release: The year a movie was released.
* Runtime: The duration of the movie.
* Genre: The genre(s) of the movie.
* Story: The plot or summary of the movie.
* Rating: IMDB rating of the movie.
* Votes: The number of votes the movie received.
* Cast: The actors starring in the movie.

Creating a New DataFrame (df):

We will create a new DataFrame df with the following columns:

* Sr.no, imdb\_id, original\_title, actors, is\_adult, year\_of\_release, runtime, genres, review.

Data Analysis and Cleaning:

1. imdb\_id:
   * Unique identifier for each movie.
   * Data type: Object.
   * No null values, 46 duplicate values (1% of the data).
   * Action: Keep the first occurrence and drop the duplicates.
2. original\_title:
   * Contains the movie name.
   * Data type: Object.
   * No null values, 237 duplicate values (5% of the data).
   * Action: Keep the first occurrence and drop the duplicates.
3. actors:
   * Contains the names of actors.
   * Data type: Object.
   * Some actor names are repeated and some are missing.
   * Action: Fill missing actors with "unknown" and keep duplicate actors as they are. Convert data into a string and replace | with , as a separator.
4. is\_adult:
   * Data type: Object, stores Boolean values ('0' = No, '1' = Yes).
   * No null values, but 4282 duplicate values (mostly '0').
   * Action: Convert this to integer and replace '0' with "Not Adult" and '1' with "Adult".
5. year\_of\_release:
   * Data type: Integer.
   * No null values, 4213 duplicate values (multiple movies released in the same year).
   * No action required as multiple movies can share the same year.
6. runtime:
   * Data type: Object, should be converted to an integer.
   * No null values, but some duplicate runtimes due to older movies having the same runtime.
   * Action: Categorize runtime into:
     + Short movie: < 1.5 hours.
     + Regular movie: 1.5 to 2.5 hours.
     + Long movie: > 2.5 hours.
   * Replace "//N" with NaN, convert values to integers, and assign the correct category.
   * Categories:
     + 3183 regular movies
     + 1068 long movies
     + 32 short movies.
7. genres:
   * Data type: Object, needs to be converted to a string.
   * No null values, but genres are repeated and separated by |.
   * Action: Replace | with , as the separator.
   * The most common genre is Drama (514 movies).
8. imdb\_rating:
   * Data type: Float.
   * No null values, but some duplicate values (many movies have the same rating).
   * Action: Categorize ratings into:
     + 0 to 5: Poor.
     + 5 to 7.5: Average.
     + 7.5 to 9: Hit.
     + 9 to 10: Blockbuster.
   * Categories:
     + 2972 average movies
     + 981 poor movies
     + 329 hit movies
     + 1 blockbuster movie.
   * Plotted graphs indicate that poor movies were more prevalent in older times, and hit movies generally have an average of 13,000 votes.
9. imdb\_votes:
   * Data type: Integer.
   * No null values, but some duplicate values due to many movies having the same number of votes.
   * Action: Categorize votes into:
     + Low votes: ≤ 1000
     + Moderate votes: ≤ 1 lakh
     + High votes: ≤ 2 lakh
     + Blockbuster votes: > 2 lakh
   * Categories:
     + 3226 low votes
     + 1051 moderate votes
     + 5 high votes
     + 1 blockbuster vote.
   * Plotted graphs show that older movies had moderate votes.

Rank Column Creation:

We will create a new column review with the following ranks:

* Rank 1: Rating: hit or blockbuster, Votes: moderate, high, or blockbuster.
* Rank 2: Rating: hit or blockbuster, Votes: low.
* Rank 3: Rating: average, Votes: moderate or high.
* Rank 4: Rating: average, Votes: low.
* Rank 5: Rating: poor.

Final DataFrame:

The final DataFrame (df) will contain the following columns:

* Sr.no, imdb\_id, original\_title, actors, is\_adult, year\_of\_release, runtime, genres, review.

The imdb\_rating and imdb\_votes columns will be dropped.

Saving the Data:

* Save the DataFrame df to a CSV file named movies\_refined.csv.
* Create a new DataFrame new\_df as a copy of df, where the review column replaces imdb\_rating and imdb\_votes.

**Creating and Preparing the new\_df DataFrame:**

* A new DataFrame named new\_df is created, containing the following columns:
  + Sr.no
  + imdb\_id
  + original\_title
  + actors
  + is\_adult
  + year\_of\_release
  + runtime
  + genres
  + review
* The columns original\_title, actors, is\_adult, year\_of\_release, runtime, genres, and review are combined into a single column named tags. This involves concatenating the values from these columns into one string.
* The columns review and year\_of\_release are of integer type, so they are converted to string before the concatenation.
* After the concatenation, the new tags column is converted into a string data type.
* The original columns (original\_title, actors, is\_adult, year\_of\_release, runtime, genres, and review) are dropped from the DataFrame, leaving only the Sr.no, imdb\_id, and tags columns.

**Text Preprocessing on the tags Column:**

* The next step is to remove punctuation from the tags column. Punctuation does not add any meaningful content to the text and removing it helps reduce unnecessary tokens.
* After removing punctuation, the tags column is tokenized, which means it is split into individual words or tokens. These tokens are stored in a new column called tokenized\_tags.
* Stop words, which are common words such as "the", "is", "and", are removed from the tokenized\_tags column. The resulting words are stored in a new column called filtered\_tags. Stop words are removed because they do not contribute significant meaning to the text.
* The remaining words in the filtered\_tags column are lemmatized, which means they are converted to their root form (for example, "running" becomes "run"). This is done to group different forms of the same word. The lemmatized words are stored in a new column called lemmatized\_tags.
* The tokenized\_tags and filtered\_tags columns are then dropped, as they are no longer needed after the lemmatization process.

**Final Outcome:**

* The final new\_df DataFrame will contain the following columns:
  + Sr.no
  + imdb\_id
  + lemmatized\_tags (which contains the processed and lemmatized tags)

**Data Type Conversion**

* The lemmatized\_tags column is converted into a string data type, stored in a new column called lemmatized\_tags\_str. This step ensures compatibility with text-based vectorization techniques, as they require inputs in string format.

**TF-IDF Vectorization**

**Why TF-IDF Is Used**

TF-IDF (Term Frequency-Inverse Document Frequency) is a text vectorization method chosen for the following reasons:

* It assigns importance to terms by evaluating their frequency within a document relative to their presence in the entire dataset.
* Produces normalized outputs, enabling direct comparison of vectors across documents.

**TF-IDF Vectorizer Configuration**  
The TfidfVectorizer function from the sklearn.feature\_extraction.text module is configured with these parameters:

* input='content': Specifies the input data is string-based content.
* encoding='utf-8': Ensures proper handling of encoded input data.
* decode\_error='replace': Replaces problematic characters with Unicode characters to prevent errors.
* use\_idf=True: Activates the inverse document frequency calculation for better weighting of terms.
* smooth\_idf=True: Avoids division by zero during the calculation of IDF for terms present in all documents.

**Vectorizing the Data**

* The lemmatized\_tags\_str column is vectorized using the configured TF-IDF vectorizer, creating a sparse matrix called tfidf\_matrix. Sparse matrices store data efficiently by representing only non-zero values, which is particularly useful for high-dimensional text data.
* The sparse matrix is then converted into a dense array using the .toarray() function, resulting in tfidf\_dense. This step is necessary for operations requiring access to all values within the vector.

**Creating the DataFrame**

* A new DataFrame, vect\_df, is created to store the dense vectors and their corresponding feature names, which represent the terms in the dataset.
* The original DataFrame is copied into a new DataFrame called merge\_df\_combined. A new column, vectorized\_tokens, is added to store the dense TF-IDF vectors row-wise. This updated DataFrame, merge\_df\_combined, is saved as a CSV file named vectorized\_df.csv.

**User Query Vectorization**

A user query is taken as input and vectorized using the same TF-IDF vectorizer configuration. The resulting sparse vector is converted into a dense array called user\_query\_vector\_dense. This vector represents the query in a high-dimensional space, enabling comparison with the document vectors stored in the vectorized\_tokens column.

**Similarity Metric**

**Why Cosine Similarity Is Chosen**

* Cosine similarity measures the cosine of the angle between two vectors. It is particularly effective for text data because:
* It evaluates the orientation (direction) of vectors rather than their magnitude.
* It highlights the relative importance of terms while ignoring absolute frequency variations, making it robust for documents of varying lengths.
* The vectorized\_tokens column from the DataFrame is prepared for similarity calculations by stacking the dense vectors into a 2D matrix, referred to as stored\_vectors. The cosine similarity between the user\_query\_vector\_dense and stored\_vectors is then computed.

**Handling Zero Similarity**

If all similarity scores are zero, indicating no significant matches between the query and the stored vectors, the following message is displayed:  
**"No matches found."**

**Selecting Top Matches**

The similarity scores are sorted in descending order to identify the top three matches. The indices of the top-scoring matches are used to fetch the corresponding rows from the DataFrame, ensuring the most relevant results are retrieved.

**Displaying Recommendations**

The top-matched documents are displayed by extracting the relevant columns from the DataFrame, such as the original tags and other metadata. The recommendations are presented in a user-friendly format, allowing users to easily interpret the results and their relevance to the query.

Practical Code :- [Basic Movie Recommender system](https://colab.research.google.com/drive/11VCH0DDfzqbu1cWqZ5rFOceO5cjEIRIC?usp=sharing)