Movie Recommendations

Zenab osama 2305289

Mariam Ahmed 2305300

Ahmed Sadek 2305355

1- data preprocessing

1. Imports Pandas for data handling.
2. Defines a function **load_data()** that:
   - Reads movie and rating data from CSV files.
   - Replaces **|** with spaces in the **genres** column (likely for text processing).
   - Drops rows with missing values in both datasets.
   - Returns the cleaned DataFrames.

This function appears to be part of a movie recommendation system where clean and structured data is essential for further processing.

```python
#1. Data Processing Scripts
import pandas as pd

def load_data():
    movies = pd.read_csv('movies.csv')
    ratings = pd.read_csv('ratings.csv')
    movies['genres'] = movies['genres'].str.replace('|', ' ')
    return movies.dropna(), ratings.dropna()
```

2- Model Implementation

```
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
```

- **numpy (np)**: Used for numerical operations (though not directly used in this snippet).
- **TfidfVectorizer**: Converts text (movie genres) into TF-IDF features (a way to represent text numerically).
- **cosine_similarity**: Computes similarity between vectors (used to find similar movies).

```python
def setup_content_model(movies):
    tfidf = TfidfVectorizer(stop_words='english')
    tfidf_matrix = tfidf.fit_transform(movies['genres'])
    return cosine_similarity(tfidf_matrix)
```

- Purpose: Build a content-based recommendation system using movie genres.
- Steps:

1. TF-IDF Vectorization:
   - Converts the genres text into a numerical matrix using TF-IDF.
   - Uses stop_words='english' to remove common words and reduce noise.
   - fit_transform learns genre terms and converts each movie's genres into a vector.
2. Cosine Similarity:
   - Calculates cosine similarity between all movie genre vectors.
   - Outputs a similarity matrix [n_movies x n_movies], where each entry [i][j] shows how similar movie i is to movie j based on genres.

- Purpose: Recommend top N movies similar in genre to a given movie.
- Inputs:
  - movies: DataFrame with movie info.
  - cosine_sim: Precomputed genre similarity matrix.
  - movie_id: ID of the target movie.
  - top_n: Number of recommendations (default = 10).
- Steps:

a. Find Movie Index:
   - Locate the row in movies where movieId matches movie_id.

b. Get Similarities:
   - Use cosine_sim to get similarity scores between the target movie and all others.

c. Sort Scores:
   - Sort all movies by similarity score (highest first).
   - Skip the first one (it's the movie itself).

d. Get Top N Indices:
   - Select the indices of the top N most similar movies.

e. Return Recommendations:
   - Use these indices to extract and return movie info (movieId, title, genres).

```python
def get_content_recommendations(movies, cosine_sim, movie_id, top_n=10):
    idx = movies.index[movies['movieId'] == movie_id].tolist()[0]
    sim_scores = list(enumerate(cosine_sim[idx]))
    sim_scores = sorted(sim_scores, key=lambda x: x[1], reverse=True)[1:top_n+1]
    indices = [i[0] for i in sim_scores]
    return movies.iloc[indices][['movieId', 'title', 'genres']]
```

```
from surprise import Dataset, Reader, SVD
import numpy as np
```

- **surprise**: A Python library for building recommender systems.
  - **Dataset**: Loads and manages rating data.
  - **Reader**: Parses rating data (e.g., scale, format).
  - **SVD**: Implements the Singular Value Decomposition algorithm (a matrix factorization technique for recommendations).
- **numpy** (**np**): Used for numerical operations (e.g., set differences).

```python
def setup_collaborative_model(ratings):
    reader = Reader(rating_scale=(0.5, 5.0))
    data = Dataset.load_from_df(ratings[['userId', 'movieId', 'rating']], reader)
    trainset = data.build_full_trainset()
    svd = SVD()
    svd.fit(trainset)
    return svd
```

- Purpose: Trains a collaborative filtering model using SVD (Singular Value Decomposition) on user-movie rating data.
- Input:
- A ratings DataFrame with columns: userId, movieId, and rating.
- Process:

1. Define Rating Scale:
   - Uses a Reader to specify the rating range (0.5 to 5.0).
2. Convert Data:
   - Transforms the DataFrame into a format compatible with the Surprise library using load_from_df.
3. Build Training Set:
   - Prepares all data for training with build_full_trainset.
4. Train Model:
   - Initializes an SVD model and trains it on the full dataset.

- Output:
- Returns the trained SVD model, ready for making user-specific rating predictions

- Purpose:
- Recommends top N movies to a specific user using a trained SVD collaborative filtering model.
- Inputs:
- movies: Movie metadata (movieId, title, genres).
- ratings: Historical user ratings.
- svd: Trained SVD model.
- user_id: ID of the user to recommend movies to.
- top_n: Number of recommendations to return (default = 10).
- Process:

1. Identify Unrated Movies:
   - Find all movies the user hasn't rated yet.
2. Prepare Test Data:
   - Create prediction input for those unrated movies (using dummy rating values).
3. Predict Ratings:
   - Use the SVD model to estimate the user's rating for each unrated movie.
4. Select Top Predictions:
   - Sort movies by predicted rating and select the top N.
5. Return Recommendations:
   - Return the movie info (ID, title, genres) for the top recommended movies

```python
def get_collab_recommendations(movies, ratings, svd, user_id, top_n=10):
    all_movie_ids = movies['movieId'].unique()
    rated = ratings[ratings['userId'] == user_id]['movieId']
    to_predict = np.setdiff1d(all_movie_ids, rated)
    testset = [[user_id, mid, 4.] for mid in to_predict]
    preds = svd.test(testset)
    top_preds = sorted(preds, key=lambda x: x.est, reverse=True)[:top_n]
    top_ids = [pred.iid for pred in top_preds]
    return movies[movies['movieId'].isin(top_ids)][['movieId', 'title', 'genres']]
```

3. Hybrid Model Integration

```python
import pandas as pd

def hybrid_recommend(movies, content_sim, svd, user_id, movie_title, content_w=0.5, collab_w=0.5):
    try:
        movie_id = movies[movies['title'] == movie_title]['movieId'].values[0]
        content = get_content_recommendations(movies, content_sim, movie_id)
        content['score'] = content_w

        collab = get_collab_recommendations(movies, ratings, svd, user_id)
        collab['score'] = collab_w

        hybrid = pd.concat([content, collab])
        return hybrid.groupby(['movieId', 'title', 'genres']).agg({'score': 'sum'}).reset_index()
    except Exception as e:
        raise Exception(f"Recommendation error: {e}")
```

- Purpose: Combines content-based and collaborative filtering into one hybrid recommendation model.
- Inputs: Movie metadata, content similarity matrix, trained SVD model, user ID, reference movie title, and model weights.
- Handles Errors: Uses try-except to catch issues like invalid movie titles.
- Content-Based Part: Finds similar movies to a given title using genre similarity and assigns a fixed score (content_w).
- Collaborative Part: Gets top recommendations for the user using SVD and assigns a fixed score (collab_w).
- Merge Results: Combines both recommendation sets using pd.concat().
- Score Aggregation: Groups by movie info and sums duplicate scores (for movies recommended by both methods).
- Weights Control Influence: You can adjust content_w and collab_w to prioritize one method over the other.
- Flexible and Robust: Works even if a movie is recommended by only one method.
- Output: Returns a ranked DataFrame of recommended movies with their total score.

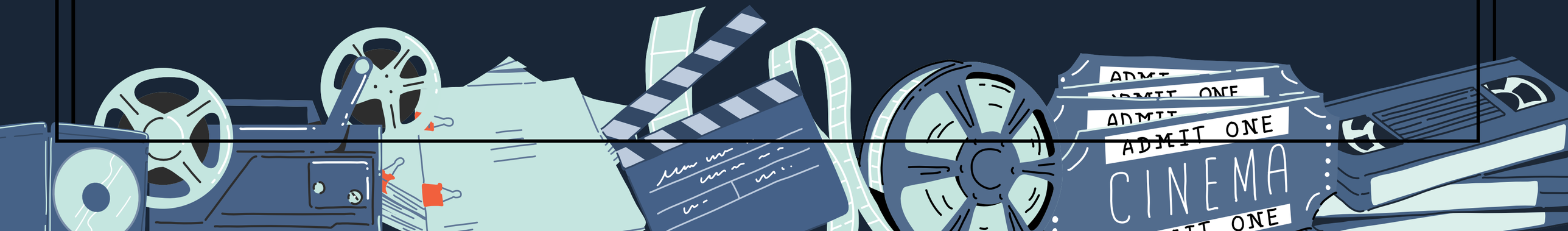
4. User Interface

4.1 imports:

- **numpy**: Used for numerical operations (though not directly visible here).
- **streamlit**: Creates the interactive web app.
- Local modules (**data_processing**, **content_based**, etc.) import the functions defined earlier.
- **warnings**: Suppresses warnings to keep the UI clean.

```python
import numpy as np
#np.import_array()  # THIS IS CRITICAL - MUST BE CALLED BEFORE SURPRISE IMPORTS

import streamlit as st
from data_processing import load_data
from content_based import setup_content_model, get_content_recommendations
from collaborative import setup_collaborative_model, get_collab_recommendations
from hybrid_model import hybrid_recommend
import warnings
warnings.filterwarnings('ignore')
```



4.2 Initialization with Error Handling

```
try:
    movies, ratings = load_data()
    cosine_sim = setup_content_model(movies)
    svd = setup_collaborative_model(ratings)
except Exception as e:
    st.error(f"Initialization failed: {str(e)}")
    st.stop()
```

- Purpose: Loads data and initializes models upfront (critical for performance).
- Error Handling:
  - If initialization fails (e.g., missing files), displays an error message and stops the app

4.3 streamlit

```
st.set_page_config(page_title="Recommender System", layout="wide")
st.title("🎬 Movie Recommender System")

app_mode = st.sidebar.radio("Choose mode", ["Home", "Content-Based", "Collaborative", "Hybrid"])
```

1. Page Configuration:
   - Sets the app title as "Recommender System".
   - Uses wide layout for better visual spacing.
2. App Title:
   - Displays a main header with a movie emoji: "🎬 Movie Recommender System".
3. Sidebar Navigation:
   - Uses radio buttons in the sidebar to let users choose between four modes:
     - Home: Shows the dataset overview.
     - Content-Based: Recommends movies based on genre similarity.
     - Collaborative: Recommends based on user ratings using collaborative filtering.
     - Hybrid: Combines both methods for improved recommendations.

```python
if app_mode == "Home":
    st.markdown("## MovieLens Dataset Overview")
    st.write(f"Total Movies: {len(movies)}")
    st.write(f"Total Ratings: {len(ratings)}")
    st.write(f"Total Users: {ratings['userId'].nunique()}")
    st.dataframe(movies.head(5))
```

- Displays:
  - Basic stats (movie count, ratings count, unique users).
  - A preview of the movies DataFrame (first 5 rows).

```
elif app_mode == "Content-Based":
    movie = st.selectbox("Choose a movie", sorted(movies['title'].unique()))
    if st.button("Recommend Similar Movies"):
        mid = movies[movies['title'] == movie]['movieId'].values[0]
        recs = get_content_recommendations(movies, cosine_sim, mid)  # Fixed missing args
        st.write(recs)
```

1. User selects a movie from a dropdown (**selectbox**).
2. Clicking the button triggers:
    - Fetching the **movieId** of the selected title.
    - Generating recommendations using **get_content_recommendations**.
    - Displaying results in a table.

```python
elif app_mode == "Collaborative":
    uid = st.number_input("Enter User ID", min_value=1, max_value=610, value=1)
    if st.button("Recommend Movies"):
        recs = get_collab_recommendations(movies, ratings, svd, uid)  # Fixed missing args
        st.write(recs)

elif app_mode == "Hybrid":
    uid = st.number_input("User ID", min_value=1, max_value=610, value=1)
    movie = st.selectbox("Your favorite movie", sorted(movies['title'].unique()))
    cw = st.slider("Content Weight", 0.0, 1.0, 0.5)
    mw = st.slider("Collaborative Weight", 0.0, 1.0, 0.5)
    if st.button("Get Hybrid Recommendations"):
        result = hybrid_recommend(movies, cosine_sim, svd, uid, movie, cw, mw)  # Fixed args
        if result is not None:
            st.write(result)
```

1. User enters their **userId** (limited to 1–610, default=1).
2. Clicking the button:
   - Calls **get_collab_recommendations** with the user ID.
   - Shows personalized recommendations.
   - User provides:
   - **userId**.
   - A favorite movie (for content-based filtering).
   - Sliders to adjust weights for content vs. collaborative (default: 0.5 each).
   - Clicking the button:
   - Calls **hybrid_recommend** to merge both recommendation types.
   - Displays results with combined scores.
   -

```python
def evaluate_model(predictions):
    """Calculate evaluation metrics"""
    rmse = accuracy.rmse(predictions)
    mae = accuracy.mae(predictions)
    return {'RMSE': rmse, 'MAE': mae}
```

- Purpose: Assess how accurate a recommendation model's predicted ratings are.
- Key Metrics:
- RMSE: Measures squared error; sensitive to large mistakes.
- MAE: Measures average error; easier to understand.
- Input: List of prediction results.
- Output: Dictionary with RMSE and MAE scores.
- Use Case: Helps evaluate collaborative filtering performance.

```python
def compute_top_n_metrics(testset, user_id, top_n=10):
    """Calculate precision/recall metrics"""
    test_df = pd.DataFrame([(pred.uid, pred.iid, pred.r_ui, pred.est) for pred in testset],
                           columns=['userId', 'movieId', 'actual', 'predicted'])
    user_df = test_df[test_df['userId'] == user_id]

    if user_df.empty:
        return None

    actual_liked = user_df[user_df['actual'] >= 4]['movieId'].values
    predicted_top = user_df.sort_values('predicted', ascending=False)['movieId'].head(top_n).values

    true_positives = len(np.intersect1d(actual_liked, predicted_top))
    precision = true_positives / top_n if top_n else 0
    recall = true_positives / len(actual_liked) if len(actual_liked) > 0 else 0
    f1 = 2 * (precision * recall) / (precision + recall) if (precision + recall) > 0 else 0

    return {
        'precision': precision,
        'recall': recall,
        'f1_score': f1,
        'actual_count': len(actual_liked),
        'recommended_count': len(predicted_top)
    }
```

- Purpose:
- Evaluates top-N recommendation quality for a specific user using precision, recall, and F1-score.
- Input:
- testset: List of prediction results.
- user_id: ID of the target user.
- top_n: Number of top recommendations to evaluate (default = 10).
- Process:

1. Converts predictions to a DataFrame.
2. Filters predictions for the target user.
3. Identifies "liked" movies (actual rating ≥ 4).
4. Gets top-N movies based on predicted scores.
5. Computes true positives, precision, recall, and F1-score.

- Output:
- A dictionary with precision, recall, F1-score, and counts of liked and recommended movies.
- Use Case:
- Best for evaluating ranking effectiveness in top-N recommender systems.

```python
from surprise import Dataset, Reader, SVD
import numpy as np

def setup_collaborative_model(ratings):
    reader = Reader(rating_scale=(0.5, 5.0))
    data = Dataset.load_from_df(ratings[['userId', 'movieId', 'rating']], reader)
    trainset = data.build_full_trainset()
```

Purpose:

Builds and trains a collaborative filtering model using user-item ratings.

Key Libraries:

Dataset: Manages data loading.

Reader: Defines rating format and scale.

SVD: Matrix factorization algorithm.

NumPy: Supports numerical operations (e.g., for filtering).

Process Overview:

A Reader object is created to define the rating scale (e.g., 0.5–5.0).

A Surprise Dataset is created from a DataFrame containing userId, movieId, and rating.

The full training set is built to prepare the data for model training.

The SVD model is initialized and trained using this complete dataset.

```
svd = SVD()
svd.fit(trainset)
return svd
```

- Initializes the SVD model with default settings.
- Trains the model on the full training dataset, learning latent user and item factors.
- Returns the trained SVD model ready for making rating predictions.

```python
def get_collab_recommendations(movies, ratings, svd, user_id, top_n=10):
    all_movie_ids = movies['movieId'].unique()
    rated = ratings[ratings['userId'] == user_id]['movieId']
    to_predict = np.setdiff1d(all_movie_ids, rated)
    testset = [[user_id, mid, 4.] for mid in to_predict]
    preds = svd.test(testset)
    top_preds = sorted(preds, key=lambda x: x.est, reverse=True)[:top_n]
    top_ids = [pred.iid for pred in top_preds]
    return movies[movies['movieId'].isin(top_ids)][['movieId', 'title', 'genres']]
```

Generates personalized recommendations by predicting ratings only for movies a user hasn’t rated. Uses SVD to rank and return the top-N movies based on predicted scores, providing efficient, user-specific suggestions.

1. Content-Based Filtering

```python
def get_collab_recommendations(movies, ratings, svd, user_id, top_n=10):
    all_movie_ids = movies['movieId'].unique()
    rated = ratings[ratings['userId'] == user_id]['movieId']
    to_predict = np.setdiff1d(all_movie_ids, rated)
    testset = [[user_id, mid, 4.] for mid in to_predict]
    preds = svd.test(testset)
    top_preds = sorted(preds, key=lambda x: x.est, reverse=True)[:top_n]
    top_ids = [pred.iid for pred in top_preds]
    return movies[movies['movieId'].isin(top_ids)][['movieId', 'title', 'genres']]
```

- enre text processing for better feature extraction
- Cached similarity matrix for fast recommendations

Collaborative filtering setup

```
@st.cache_resource
def setup_collaborative():
    reader = Reader(rating_scale=(0.5, 5.0))
    data = Dataset.load_from_df(ratings[['userId', 'movieId', 'rating']], reader)
    trainset, testset = train_test_split(data, test_size=0.2, random_state=42)
    svd = SVD()
    svd.fit(trainset)
    predictions = svd.test(testset)
    return svd, predictions, testset

svd, predictions, testset = setup_collaborative()
```

- Proper train-test split for evaluation
- SVD implementation for latent factor modeling

```python
def get_content_recommendations(movie_id, top_n=15):
    idx = movies.index[movies['movieId'] == movie_id].tolist()[0]
    sim_scores = list(enumerate(cosine_sim[idx]))
    sim_scores = sorted(sim_scores, key=lambda x: x[1], reverse=True)[1:top_n+1]
    indices = [i[0] for i in sim_scores]
    return movies.iloc[indices][['movieId', 'title', 'genres']]
```

- Takes a movie_id and returns top_n similar movies.
- Finds the index of the movie in the dataset.
- Computes cosine similarity scores for that movie against others.
- Sorts and returns the most similar movies (excluding itself).

```python
def get_collab_recommendations(user_id, top_n=15):
    all_movie_ids = movies['movieId'].unique()
    rated = ratings[ratings['userId'] == user_id]['movieId']
    to_predict = np.setdiff1d(all_movie_ids, rated)
    testset = [[user_id, mid, 4.] for mid in to_predict]
    preds = svd.test(testset)
    top_preds = sorted(preds, key=lambda x: x.est, reverse=True)[:top_n]
    top_ids = [pred.iid for pred in top_preds]
    return movies[movies['movieId'].isin(top_ids)][['movieId', 'title', 'genres']]
```

- Recommends movies for a given user_id using the SVD model.
- Identifies movies the user hasn't rated.
- Predicts ratings for unseen movies and returns the top top_n highest-rated ones.

```python
def hybrid_recommend(user_id, movie_title, content_w=0.5, collab_w=0.5):
    try:
        movie_id = movies[movies['title'] == movie_title]['movieId'].values[0]
        content = get_content_recommendations(movie_id)
        content['score'] = content_w

        collab = get_collab_recommendations(user_id)
        collab['score'] = collab_w

        hybrid = pd.concat([content, collab])
        hybrid = hybrid.groupby(['movieId', 'title', 'genres']).agg({'score': 'sum'}).reset_index()
        return hybrid.sort_values('score', ascending=False).head(16)
    except Exception as e:
        st.error(f"Recommendation error: {e}")
        return None
```

- Combines content-based and collaborative recommendations.
- Weights each method's contribution (content_w and collab_w).
- Merges and ranks recommendations by aggregated scores.

```python
def compute_top_n_metrics(user_id, top_n=15):
    test_df = pd.DataFrame([(pred.uid, pred.iid, pred.r_ui, pred.est) for pred in predictions],
                           columns=['userId', 'movieId', 'actual', 'predicted'])
    user_df = test_df[test_df['userId'] == user_id]

    if user_df.empty:
        return None

    actual_liked = user_df[user_df['actual'] >= 4]['movieId'].values
    predicted_top = user_df.sort_values('predicted', ascending=False)['movieId'].head(top_n).values

    true_positives = len(np.intersect1d(actual_liked, predicted_top))
    precision = true_positives / top_n if top_n else 0
    recall = true_positives / len(actual_liked) if len(actual_liked) > 0 else 0
    f1 = 2 * (precision * recall) / (precision + recall) if (precision + recall) > 0 else 0

    return {
        'precision': precision,
        'recall': recall,
        'f1_score': f1,
        'actual_count': len(actual_liked),
        'recommended_count': len(predicted_top)
    }
```

- Computes precision, recall, and F1-score for a user's recommendations.
- Compares predicted top movies against movies the user actually liked (rated ≥ 4).
- Returns metrics to assess recommendation quality.

```python
# Streamlit UI
st.title("🎬 Hybrid Movie Recommendation System")

app_mode = st.sidebar.radio("Choose mode", ["Home", "Content-Based", "Collaborative", "Hybrid", "Evaluation"])

if app_mode == "Home":
    st.markdown("## MovieLens Dataset Overview")
    st.write(f"Total Movies: {len(movies)}")
    st.write(f"Total Ratings: {len(ratings)}")
    st.write(f"Total Users: {ratings['userId'].nunique()}")
    st.dataframe(movies.head(5))
elif app_mode == "Content-Based":
    movie = st.selectbox("Choose a movie", sorted(movies['title'].unique()))
    if st.button("Recommend Similar Movies"):
        mid = movies[movies['title'] == movie]['movieId'].values[0]
        recs = get_content_recommendations(mid)
        st.write(recs)
elif app_mode == "Collaborative":
    uid = st.number_input("Enter User ID", min_value=1, max_value=610, value=1)
    if st.button("Recommend Movies"):
        st.write(get_collab_recommendations(uid))
elif app_mode == "Hybrid":
    uid = st.number_input("User ID", min_value=1, max_value=610, value=1)
    movie = st.selectbox("Your favorite movie", sorted(movies['title'].unique()))
    cw = st.slider("Content Weight", 0.0, 1.0, 0.5)
    mw = st.slider("Collaborative Weight", 0.0, 1.0, 0.5)
    if st.button("Get Hybrid Recommendations"):
        result = hybrid_recommend(uid, movie, cw, mw)
        if result is not None:
            st.write(result)
elif app_mode == "Evaluation":
```

- Home: Displays dataset stats (movies, ratings, users).
- Content-Based: Recommends similar movies based on a selected movie.
- Collaborative: Recommends movies for a user based on their past ratings.
- Hybrid: Blends both methods with adjustable weights.
- Evaluation: Shows model performance (RMSE, MAE) and user-specific metrics.

```python
elif app_mode == "Evaluation":
    st.subheader("Model Performance")
    st.write(f"RMSE: {accuracy.rmse(predictions):.4f}")
    st.write(f"MAE: {accuracy.mae(predictions):.4f}")

    uid = st.number_input("Evaluate for User ID:", min_value=1, max_value=610, value=1)
    if st.button("Evaluate Top-N Precision/Recall"):
        metrics = compute_top_n_metrics(uid)
        if metrics:
            st.write(f"Precision@10: {metrics['precision']:.2f}")
            st.write(f"Recall@10: {metrics['recall']:.2f}")
            st.write(f"F1 Score: {metrics['f1_score']:.2f}")
            st.write(f"Actual liked movies: {metrics['actual_count']}")
            st.write(f"Movies recommended: {metrics['recommended_count']}")
        else:
            st.warning("This user has no valid ratings in the test set.")
st.warning("Modifed By Sadek.")
```