Recommendation Systems

Content-Based, Collaborative, and Hybrid Approaches

Content-Based Recommendation System

Content:

- Focuses on the characteristics (content) of items.
- Builds a profile for each item based on its features (e.g., genres, keywords, actors, directors).
- Builds a profile for the user based on the features of items they have interacted with (e.g., liked, rated highly).
- Recommends items whose content profile matches the user's profile.
- In this implementation, we use movie metadata (genres and tags) to create a content representation for each movie.

Data Loading - Content-Based System

Content:

Loading the datasets required for building the content profiles and recommendations.

Datasets include:

- Movie.csv: contains movie ids, titles, and geners
- Tag.csv: user-submitted tags for movies.
- Genome_scores.csv: Relevance scores between movies and genome tags.
- Genome_tags.csv: Genome tag IDs their corresponding tag text.

First Cell Explanation

```
import pandas as pd
import numpy as np
import ast
From sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.metrics import mean_squared_error, mean_absolute_error
from sklearn.metrics import precision_score, recall_score, f1_score,
roc_auc_score, roc_curve, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns

movies = pd.read_csv('movie.csv')
tags = pd.read_csv('tag.csv')
genome_scores = pd.read_csv('genome_scores.csv')
genome_tags = pd.read_csv('genome_tags.csv')
```

Explanation

- This block imports the necessary Python libraries.
  - pandas: For data manipulation and analysis, primarily using DataFrames.
  - numpy: For numerical operations, especially array manipulation.
  - ast: Used later for safely evaluating strings containing Python literal structures (though not strictly needed for the provided functions).
  - sklearn.feature_extraction.text.TfidfVectorizer: To convert text data (movie genres and tags) into numerical feature vectors using the TF-IDF method.
  - sklearn.metrics.pairwise.cosine_similarity: To calculate the similarity between the TF-IDF vectors.
  - sklearn.metrics modules: For evaluating the recommendation system's performance using various metrics (Mean Squared Error, Mean Absolute Error, Precision, Recall, F1-Score, AUC, ROC curve, Confusion Matrix).
  - matplotlib.pyplot: For creating static, interactive, and animated visualizations (plots).
  - seaborn: A statistical data visualization library based on matplotlib, used for creating aesthetically pleasing plots like bar charts and heatmaps.
- The last four lines use pd.read_csv() to load the data from CSV files into pandas DataFrames named movies, tags, genome_scores, and genome_tags. These DataFrames will be used throughout the script.

Second cell explanation

```
print("Movies Count:")
print(movies.count())
print("\nTags Count:")
print(tags.count())
print("\nGenome scores Count:")
print(genome_scores.count())
print("\nGenome Tags Count:")
print(genome_tags.count())
```

Explanation

- This snippet prints the non-null counts for each column in the four loaded .
- DataFrames (movies, tags, genome_scores, genome_tags).
- The .count() method on a DataFrame returns a Series containing the count of non-null values for each column.
- This is a basic data exploration step to quickly see if there are significant missing values in any columns right after loading the data. print statements are used to display these counts with descriptive headers.

Third Cell explanation

```
  genome = pd.merge(genome_scores, genome_tags, on='tagId')

  genome_sorted = genome.sort_values(['movieId', 'relevance'], ascending=[True, False])

  top_genome_tags = genome_sorted.groupby('movieId').head(5)

  top_genome_tags_grouped = top_genome_tags.groupby('movieId')['tag'].apply(list).reset_index()

  top_genome_tags_grouped.rename(columns={'tag': 'genome_tags'}, inplace=True)


user_tags_grouped = tags.groupby('movieId')['tag'].apply(list).reset_index()

  user_tags_grouped.rename(columns={'tag': 'user_tags'}, inplace=True)
```

Explanation

- genome = pd.merge(genome_scores, genome_tags, on='tagId'): Merges the genome_scores DataFrame with the genome_tags DataFrame using the common column tagId. This combines the relevance scores with the actual tag text.
- genome_sorted = genome.sort_values(['movieId', 'relevance'], ascending=[True, False]): Sorts the merged genome DataFrame first by movieId in ascending order, and then by relevance score in descending order. This brings the most relevant genome tags to the top for each movie.
- top_genome_tags = genome_sorted.groupby('movieId').head(5): Groups the sorted data by movieId and selects the first 5 rows for each group (which are the top 5 most relevant tags due to the previous sorting).
- top_genome_tags_grouped = top_genome_tags.groupby('movieId')['tag'].apply(list).reset_index(): Groups the top_genome_tags by movieId and aggregates the tag column into a list for each movie. .reset_index() converts the grouped output back into a DataFrame.
- top_genome_tags_grouped.rename(columns={'tag': 'genome_tags'}, inplace=True): Renames the aggregated tag column from 'tag' to 'genome_tags' for clarity. inplace=True modifies the DataFrame directly.
- user_tags_grouped = tags.groupby('movieId')['tag'].apply(list).reset_index(): Groups the original tags DataFrame by movieId and aggregates all user-submitted tags into a list for each movie.
- user_tags_grouped.rename(columns={'tag': 'user_tags'}, inplace=True): Renames the aggregated user tag column to 'user_tags'.

Fourth Cell Explanation

```
movies =
movies.merge(top_genome_tags
_grouped, on='movieId',
how='left’)

movies =
movies.merge(user_tags_group
ed, on='movieId',
how='left’)
```

Explanation

- movies = movies.merge(top_genome_tags_grouped, on='movieId', how='left'): Merges the movies DataFrame with the top_genome_tags_grouped DataFrame. The merge is done on the common movieId column. A how='left' merge ensures that all movies from the original movies DataFrame are kept, and the genome tags are added where a match exists. Movies without genome tags will have NaN in the new 'genome_tags' column.
- movies = movies.merge(user_tags_grouped, on='movieId', how='left'): Similarly, merges the result with the user_tags_grouped DataFrame on movieId using a left merge. Movies without user tags will have NaN in the new 'user_tags' column. After these merges, the movies DataFrame now contains columns for movie details, processed genome tags, and processed user tags.

Fifth Cell explanation

```
def process_genres(genres):
    try:
        return genres.lower().replace('|', ' ').split()
    except:
        return []

def clean_tags(tag_list):
    try:
        return [str(tag).lower().replace(' ', '') for tag in tag_list if isinstance(tag, str)]
    except:
        return []
```

Explanation

process_genres(genres) function:

- Takes a string genres as input (e.g., "Adventure|Animation|Children").
- It's wrapped in a try...except block to handle potential errors, returning an empty list [] if an error occurs (e.g., if genres is not a string).
- Inside the try block:
  - genres.lower(): Converts the genre string to lowercase (e.g., "adventure|animation|children").
  - .replace('|', ' '): Replaces the pipe symbol | with a space (e.g., "adventure animation children").
  - .split(): Splits the string into a list of words based on spaces (e.g., ['adventure', 'animation', 'children']).
- This function transforms the genre string into a standardized list of lowercase genre terms.

- **clean_tags(tag_list) function:**
  - Takes a list tag_list as input (e.g., ['action', 'super hero', nan]).
  - It's wrapped in a try...except block, returning [] if an error occurs (e.g., if tag_list is not iterable or contains unexpected data types causing issues).
  - Inside the try block, it uses a list comprehension:
    - for tag in tag_list: Iterates through each item in the input list.
    - if isinstance(tag, str): Checks if the current item tag is a string. This prevents errors if the list contains non-string values like NaN which can occur from left merges.
    - str(tag): Converts the tag to a string. This is a safety measure, though the isinstance check makes it less critical.
    - .lower(): Converts the tag string to lowercase.
    - .replace(' ', ''): Removes any spaces within the tag text (e.g., "super hero" becomes "superhero"). This is done to treat multi-word tags as single terms.
  - This function cleans and standardizes a list of tags into a list of lowercase, space-removed strings, filtering out non-string entries.

Sixth Cell explanation

```
  movies['genres'] =
movies['genres'].apply(process_genres)

  movies['genome_tags'] =
movies['genome_tags'].apply(clean_tags)

  movies['user_tags'] =
movies['user_tags'].apply(clean_tags)


movies['tags'] = movies['genres'] +
movies['genome_tags'] + movies['user_tags']

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

Explanation

These lines use the .apply() method on pandas Series (DataFrame columns) to execute the previously defined functions on each element of the respective columns.

movies['genres'].apply(process_genres): Applies the process_genres function to every value in the 'genres' column. The original string format like "Action|Adventure" is replaced by a list of processed strings like ['action', 'adventure'].

movies['genome_tags'].apply(clean_tags): Applies the clean_tags function to every list in the 'genome_tags' column. Each list of genome tags is cleaned (lowercased, spaces removed, non-strings handled).

- movies['user_tags'].apply(clean_tags): Applies the clean_tags function to every list in the 'user_tags' column. Each list of user tags is cleaned similarly.
- The results of the .apply() operations overwrite the original columns with the cleaned data.

Explanation

- movies['tags'] = movies['genres'] + movies['genome_tags'] + movies['user_tags']: Creates a new column named 'tags'. For each movie, it concatenates the lists of strings from the 'genres', 'genome_tags', and 'user_tags' columns. If a movie has NaN in 'genome_tags' or 'user_tags' after the left merge, the clean_tags function returns an empty list [], so the concatenation works correctly without errors. The result is a single list of all relevant descriptive terms for each movie.
- movies['tags'] = movies['tags'].apply(lambda x: ' '.join(x)): Applies a lambda function to each list in the new 'tags' column. The lambda x: ' '.join(x) takes a list x (the combined tags) and joins its elements into a single string with spaces in between. This creates a space-separated "document" of all content terms for each movie, suitable for text vectorization.

Seventh Cell explanation

```
tfidf = TfidfVectorizer(max_features=5000, stop_words='english')

tfidf_matrix = tfidf.fit_transform(movies['tags'])


similarity = cosine_similarity(tfidf_matrix)
```

Explanation

tfidf = TfidfVectorizer(max_features=5000, stop_words='english'): Initializes a TfidfVectorizer object.

max_features=5000: Limits the vocabulary size to the top 5000 most frequent terms after ignoring stop words. This reduces the dimensionality of the resulting vectors.

- stop_words='english': Instructs the vectorizer to remove common English words (like 'the', 'a', 'is') before calculating TF-IDF. These words are usually not discriminative for content similarity.
- tfidf_matrix = tfidf.fit_transform(movies['tags']): This is a two-step process:
- tfidf.fit(movies['tags']): The vectorizer learns the vocabulary and the inverse document frequencies from the 'tags' column of all movies.
- tfidf.transform(movies['tags']): Transforms each movie's 'tags' string into a sparse TF-IDF matrix representation. Each row corresponds to a movie, and each column corresponds to a unique term in the vocabulary, with values representing the TF-IDF score of that term in that movie's content. The result tfidf_matrix is a sparse matrix, which is memory-efficient for high-dimensional data with many zero values.

Explanation

similarity = cosine_similarity(tfidf_matrix): Calculates the cosine similarity matrix.

It takes the tfidf_matrix as input. The result is a square matrix where similarity[i][j] represents the cosine similarity between the TF-IDF vector of movie i and the TF-IDF vector of movie j.

This matrix is symmetric (similarity[i][j] equals similarity[j][i]). The diagonal elements (similarity[i][i]) will be 1, as a movie is perfectly similar to itself.

This similarity matrix is the core component of the content-based recommender, storing the similarity scores between all pairs of items based on their content.

Eighth Cell explanation

```
  def recommend(title, top_n=5):
      if title not in movies['title'].values:
          return "Movie not found."

    idx = movies[movies['title'] == title].index[0]
      distances = list(enumerate(similarity[idx]))
      sorted_movies = sorted(distances, key=lambda x: x[1], reverse=True)[1:top_n+1]

    recommended_titles = [movies.iloc[i[0]]['title'] for i in sorted_movies]
      return recommended_titles

print(recommend("Iron Man (2008)"))

def is_relevant(source_idx, rec_idx):
      source_genres = set(movies.iloc[source_idx]['genres'])
      rec_genres = set(movies.iloc[rec_idx]['genres'])
      return int(len(source_genres & rec_genres) > 0)
```

Explanation

- **def recommend(title, top_n=5)::** Defines a function named recommend that takes title (the movie title to get recommendations for) and top_n (the number of recommendations to return, defaulting to 5) as arguments.

- if title not in movies['title'].values:: Checks if the input title exists in the 'title' column of the movies DataFrame. If not, it returns "Movie not found."

- idx = movies[movies['title'] == title].index[0]: If the movie is found, this line gets the index of that movie in the movies DataFrame. movies['title'] == title creates a boolean Series, movies[...] filters the DataFrame, .index gets the index object, and [0] takes the first index (assuming unique titles, or the first match).

- distances = list(enumerate(similarity[idx])): Retrieves the row corresponding to the input movie's index (idx) from the similarity matrix. This row contains the similarity scores between the input movie and *all* other movies. enumerate() pairs each score with its original index [(0, score_0), (1, score_1), ..., (N, score_N)]. This list is converted to a list.

- sorted_movies = sorted(distances, key=lambda x: x[1], reverse=True)[1:top_n+1]: Sorts the distances list.

- key=lambda x: x[1]: Sorts based on the second element of each tuple (x[1]), which is the similarity score.

Explanation

- reverse=True: Sorts in descending order (highest similarity first).
- [1:top_n+1]: Slices the sorted list. [1:] skips the first element, which is the input movie itself (similarity 1.0). :top_n+1 takes the next top_n elements.
- recommended_titles = [movies.iloc[i[0]]['title'] for i in sorted_movies]: Uses a list comprehension to extract the titles of the recommended movies.
- for i in sorted_movies: Iterates through the top N tuples (index, score).
- i[0]: Gets the original index of the recommended movie.
- movies.iloc[i[0]]['title']: Uses the original index (iloc) to retrieve the row from the movies DataFrame and then extracts the 'title' from that row.
- return recommended_titles: Returns the list of recommended movie titles.

Explanation

- **def is_relevant(source_idx, rec_idx)::** Defines a function to determine if a target movie (rec_idx) is "relevant" to a source movie (source_idx).
- source_genres = set(movies.iloc[source_idx]['genres']): Retrieves the list of genres for the source movie using its index (source_idx) and converts it into a set. Using a set allows for efficient checking of common elements.
- rec_genres = set(movies.iloc[rec_idx]['genres']): Retrieves the list of genres for the target movie (rec_idx) and converts it into a set.
- source_genres & rec_genres: Performs a set intersection operation. This returns a new set containing only the elements (genres) that are present in *both* sets.
- len(...) > 0: Checks if the resulting intersection set is not empty, meaning they share at least one genre.
- int(...): Converts the boolean result (True or False) to an integer (1 or 0). 1 indicates relevance (shared genre), and 0 indicates non-relevance (no shared genres).
- This function serves as a binary "ground truth" for evaluating the content similarity scores – a pair of movies is considered relevant if they share a genre, and not relevant otherwise. *Note: This is a very basic relevance definition and might not perfectly reflect true user preference similarity.*

Collaborative-Based Recommendation System

Content:

- Focuses on user behavior and preferences, not item content.
- **User-Based:** Recommends items that users *similar* to the active user have liked. Similarity is based on overlapping rating history.
- **Item-Based:** Recommends items that are *similar* to items the active user has liked. Item similarity is based on how other users have rated those items.
- **Matrix Factorization (e.g., SVD):** Learns latent factors (hidden characteristics) for both users and items based on the rating matrix, then predicts missing ratings. This is the primary method used in this script.
- Assumes users who agreed in the past will agree in the future.

Data Loading - Collaborative-Based System

Content:

- Loading the ratings.csv and movies.csv datasets from the specified Google Drive path.
- These datasets contain user ratings for movies and movie metadata (ID, title, genres).
- Ratings data is the core for collaborative filtering.

First Cell Explanation

```
import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

from collections import defaultdict

from surprise import SVD, Dataset, Reader, accuracy

from surprise.model_selection import train_test_split,
RandomizedSearchCV

from sklearn.metrics.pairwise import cosine_similarity

from sklearn.metrics import (

    mean_squared_error, mean_absolute_error,

    precision_score, recall_score, f1_score,

    roc_auc_score, roc_curve, confusion_matrix

)


ratings = pd.read_csv('/content/drive/My Drive/Datasets/ratings.csv')

movies = pd.read_csv('/content/drive/My Drive/Datasets/movies.csv')
```

Explanation

- This block imports the libraries needed for the collaborative filtering script. It includes pandas, numpy, plotting libraries (matplotlib.pyplot, seaborn), defaultdict (though not strictly used in the provided snippets), key modules from surprise (SVD model, data handling, accuracy metrics, model selection), and sklearn.metrics for evaluation (though Surprise has its own evaluation tools, sklearn metrics are used for classification-focused evaluation).

- ratings = pd.read_csv(...): Loads the ratings.csv file into a pandas DataFrame named ratings. This file typically contains columns like userId, movieId, rating, and timestamp.

- movies = pd.read_csv(...): Loads the movies.csv file into a pandas DataFrame named movies. This file typically contains columns like movieId, title, and genres. The path /content/drive/My Drive/Datasets/ is the location within the mounted Google Drive.

Second Cell Explanation

```
ratings.drop_duplicates(subset=['userId',
'movieId'], inplace=True)

ratings = ratings.drop('timestamp', axis=1)
```

Explanation

- ratings.drop_duplicates(subset=['userId', 'movieId'], inplace=True): Identifies and removes rows from the ratings DataFrame where the combination of 'userId' and 'movieId' is duplicated. subset=['userId', 'movieId'] specifies the columns to consider for identifying duplicates. inplace=True modifies the DataFrame directly without returning a new one. This ensures that each user has at most one rating per movie.
- ratings = ratings.drop('timestamp', axis=1): Removes the 'timestamp' column from the ratings DataFrame. axis=1 indicates that a column is being dropped. The SVD algorithm primarily uses user, item, and rating, so the timestamp is redundant for this specific model.

Third Cell Explanation

```
min_user_ratings = 10

min_movie_ratings = 10


filter_users = ratings['userId'].value_counts() >=
min_user_ratings

filter_users =
filter_users[filter_users].index.tolist()


filter_movies = ratings['movieId'].value_counts()
>= min_movie_ratings

filter_movies =
filter_movies[filter_movies].index.tolist()
```

Explanation

- min_user_ratings = 10: Sets a threshold of 10 ratings for users. Users with fewer than 10 ratings will be excluded.
- min_movie_ratings = 10: Sets a threshold of 10 ratings for movies. Movies with fewer than 10 ratings will be excluded.
- ratings['userId'].value_counts(): Calculates the number of ratings for each unique userId, returning a Series where the index is the userId and the value is the count.
- >= min_user_ratings: Creates a boolean Series, marking True for users who meet or exceed the minimum rating count.
- filter_users = filter_users[filter_users].index.tolist(): Filters this boolean Series to keep only the True entries. The .index of the resulting Series gives the userIds that meet the criterion. .tolist() converts these user IDs into a Python list. filter_users now contains the IDs of users who have rated at least min_user_ratings movies.
- The steps for filter_movies are analogous, identifying the movieIds that have received at least min_movie_ratings.

Fourth Cell Explanation

```
print(f"Before filtering: {ratings['userId'].nunique()} users, "
      f"{ratings['movieId'].nunique()} movies, {len(ratings)} ratings")


ratings = ratings[
    (ratings['userId'].isin(filter_users)) &
    (ratings['movieId'].isin(filter_movies))
].sample(n=10_000_000, random_state=42).reset_index(drop=True)

print(f"\nAfter filtering: {ratings['userId'].nunique()} users, "
      f"{ratings['movieId'].nunique()} movies, {len(ratings)} ratings")
```

Explanation

- The first print statement shows the number of unique users, movies, and total ratings *before* applying the filters, using .nunique() for unique counts and len() for total ratings.
- ratings = ratings[...]: This line filters the ratings DataFrame.
- ratings['userId'].isin(filter_users): Creates a boolean Series, True for rows where the userId is in the filter_users list.
- ratings['movieId'].isin(filter_movies): Creates a boolean Series, True for rows where the movieId is in the filter_movies list.
- (...) & (...): Combines the two boolean Series using the logical AND operator (&). A row is kept only if its userId is in filter_users AND its movieId is in filter_movies.
- ratings[...]: Filters the DataFrame using the combined boolean Series.
- .sample(n=10_000_000, random_state=42): Takes a random sample of 10 million rows from the filtered DataFrame. random_state=42 ensures the sampling is reproducible. *This step significantly reduces the size if the original filtered data is much larger than 10 million ratings.*
- .reset_index(drop=True): Resets the DataFrame index after sampling. drop=True prevents the old index from being added as a new column.
- The second print statement shows the counts *after* filtering and sampling, demonstrating the effect of these steps on the dataset size

Fifth Cell Explanation

```
reader = Reader(rating_scale=(0.5, 5))

data = Dataset.load_from_df(ratings[['userId', 'movieId',
'rating']], reader)


trainset, testset = train_test_split(data, test_size=0.2,
random_state=42)
```

Explanation

- reader = Reader(rating_scale=(0.5, 5)): Initializes a Reader object. The rating_scale parameter tells Surprise the minimum and maximum possible rating values in the dataset. This is essential for the algorithms to correctly interpret the ratings. Here, the ratings range from 0.5 to 5.0.
- data = Dataset.load_from_df(ratings[['userId', 'movieId', 'rating']], reader): Creates a Surprise Dataset object.
- ratings[['userId', 'movieId', 'rating']]: Selects only the necessary columns from the pandas ratings DataFrame. Surprise expects the data in the format of [user, item, rating].
- The selected DataFrame and the configured reader are passed to Dataset.load_from_df(). This function reads the data from the pandas DataFrame and stores it in Surprise's internal format, ready for splitting and training.

Sixth Cell Explanation

```
class SVDWithHistory(SVD):
    def fit(self, trainset):
        super().fit(trainset)
        self.train_losses = []
        for epoch in range(self.n_epochs):
            loss = 0
            for uid, iid, true_r in
trainset.all_ratings():
                est = self.predict(uid, iid,
verbose=False).est
                loss += (true_r - est) ** 2
            self.train_losses.append(loss /
trainset.n_ratings)
        return self

model = SVDWithHistory(n_factors=100, n_epochs=20,
lr_all=0.005, reg_all=0.02, random_state=42)
model.fit(trainset)
```

Explanation

- **class SVDWithHistory(SVD)::** Defines a new class SVDWithHistory that inherits from the base surprise.SVD class. This allows us to potentially add or modify behavior of the standard SVD algorithm.
- **def fit(self, trainset)::** Overrides the standard fit method from the parent SVD class. This method is called to train the model.
- super().fit(trainset): This is the crucial line. It calls the original fit method of the parent SVD class. This is where the actual SVD matrix factorization training happens (iterative optimization like stochastic gradient descent) for n_epochs. The latent factor matrices (pu, qi, bu, bi) are learned here.
- self.train_losses = []: Initializes an empty list to store calculated training loss values.
- for epoch in range(self.n_epochs):: This loop runs n_epochs times *after* the super().fit(trainset) call has completed the entire training process. It does **not** correspond to tracking loss *during* each training epoch.
- loss = 0: Initializes a variable to accumulate squared error for the current "epoch" calculation.
- for uid, iid, true_r in trainset.all_ratings():: Iterates through every single rating in the trainset.
- est = self.predict(uid, iid, verbose=False).est: Uses the *already trained* model (self, which is the result of super().fit()) to predict the rating (est) for the current user-item pair from the training set.
- loss += (true_r - est) ** 2: Calculates the squared difference between the true rating (true_r) and the predicted rating (est) and adds it to the total loss.
- self.train_losses.append(loss / trainset.n_ratings): Calculates the average squared error for this pass through the training set (loss divided by the total number of ratings in the trainset) and appends it to the train_losses list.
- return self: Returns the trained model instance.

Seventh Cell Explanation

```
def get_similar_users(user_id, model, n=5):
    try:
        user_inner_id = model.trainset.to_inner_uid(user_id)
    except ValueError:
        return []

    user_factor = model.pu[user_inner_id]
    all_user_factors = model.pu

    similarities = cosine_similarity([user_factor], all_user_factors)[0]
    sorted_indices = np.argsort(similarities)[::-1][1:n+1]

    return [(model.trainset.to_raw_uid(i), similarities[i])
            for i in sorted_indices]

similar_users = get_similar_users(7046, model)
```

Explanation

- **def get_similar_users(user_id, model, n=5)::** Defines a function to find n users similar to user_id using the trained model.
- try...except ValueError: Attempts to convert the raw user_id (from the input data) to the internal user ID used by the Surprise trainset. If the user_id was not part of the training data, it raises a ValueError, and the function returns an empty list.
- user_inner_id = model.trainset.to_inner_uid(user_id): Converts the raw user_id to the model's internal user ID.
- user_factor = model.pu[user_inner_id]: Accesses the user latent factor matrix (model.pu) which is learned during SVD training. It retrieves the latent factor vector for the target user using their internal ID.
- all_user_factors = model.pu: Gets the entire user latent factor matrix.
- cosine_similarity([user_factor], all_user_factors)[0]: Calculates the cosine similarity between the target user's latent vector (passed as a list [user_factor] because cosine_similarity expects 2D input) and all other user latent vectors. The result is a 1D array of similarity scores, and [0] extracts this array.
- np.argsort(similarities): Returns the indices that would sort the similarities array in ascending order.
- [::-1]: Reverses the indices to get them in descending order (most similar first).
- [1:n+1]: Slices the reversed indices to get the top n indices, excluding the first one (which is the user themselves, with similarity 1.0).
- return [(model.trainset.to_raw_uid(i), similarities[i]) for i in sorted_indices]: Uses a list comprehension to create a list of tuples. For each inner index i in sorted_indices, it converts it back to the raw user ID using model.trainset.to_raw_uid(i) and pairs it with its corresponding similarity score similarities[i].

Eighth Cell Explanation

```
def get_top_recommendations(user_id, model, movies_df, n=10):
    all_movie_ids = ratings['movieId'].unique()

    user_rated_movies = ratings[ratings['userId'] == user_id]['movieId']

    unseen_movies = [movie for movie in all_movie_ids if movie not in user_rated_movies.values]

    testset = [[user_id, movie_id, 0.] for movie_id in unseen_movies]
    predictions = model.test(testset)

    top_predictions = sorted(predictions, key=lambda x: x.est, reverse=True)[:n]

    top_movie_ids = [pred.iid for pred in top_predictions]
    top_ratings = [pred.est for pred in top_predictions]
    recommended_movies = movies_df[movies_df['movieId'].isin(top_movie_ids)]
    recommended_movies = recommended_movies.copy()
    recommended_movies['estimated_rating'] = top_ratings

    return recommended_movies.sort_values('estimated_rating', ascending=False)
```

Explanation

- **def get_top_recommendations(user_id, model, movies_df, n=10)::** Defines the function to get top n recommendations for user_id using model and the movies_df for movie details.
- all_movie_ids = ratings['movieId'].unique(): Gets all movie IDs that were part of the filtered dataset used for training/testing.
- user_rated_movies = ratings[ratings['userId'] == user_id]['movieId']: Filters the ratings DataFrame to find all movies the target user_id has rated.
- unseen_movies = [movie for movie in all_movie_ids if movie not in user_rated_movies.values]: Creates a list of movie IDs from all_movie_ids that are *not* present in the list of user_rated_movies. These are the movies the user hasn't seen (or rated) yet in the training data.
- testset = [[user_id, movie_id, 0.] for movie_id in unseen_movies]: Creates a list of tuples in the format expected by model.test(). For each unseen_movie_id, it creates a tuple [user_id, movie_id, 0.]. The 0. is a dummy rating required by the test method's signature but is ignored when predicting.
- predictions = model.test(testset): Uses the trained model to predict the rating for each user-unseen movie pair in the testset.
- top_predictions = sorted(predictions, key=lambda x: x.est, reverse=True)[:n]: Sorts the list of predictions based on the estimated rating (x.est) in descending order and selects the top n predictions.
- top_movie_ids = [pred.iid for pred in top_predictions] and top_ratings = [pred.est for pred in top_predictions]: Extracts the movie IDs (iid) and estimated ratings (est) from the top_predictions list into separate lists.
- recommended_movies = movies_df[movies_df['movieId'].isin(top_movie_ids)]: Filters the movies_df DataFrame to get the rows corresponding to the top_movie_ids.
- recommended_movies = recommended_movies.copy(): Creates a copy of the filtered DataFrame to prevent potential SettingWithCopyWarning issues when adding a new column.
- rating_map = {pred.iid: pred.est for pred in top_predictions} and recommended_movies['estimated_rating'] = recommended_movies['movieId'].map(rating_map): Creates a dictionary mapping movie IDs to their estimated ratings from the top_predictions. Then, uses the .map() method on the 'movieId' column of the recommended_movies DataFrame to add the corresponding 'estimated_rating'.
- return recommended_movies.sort_values('estimated_rating', ascending=False): Returns the DataFrame of recommended movies, sorted by the estimated rating in descending order.

Ninth Cell Explanation

```
user_id = 7046

user_recommendations = get_top_recommendations(user_id, model, movies)

print(f"\nTop 10 Recommendations for User {user_id}:")

print(user_recommendations[['title', 'genres', 'estimated_rating']])
```

Explanation

- user_id = 7046: Sets the specific user ID for whom recommendations are requested.
- user_recommendations = get_top_recommendations(user_id, model, movies): Calls the get_top_recommendations function with the specified user ID, the trained model, and the movies DataFrame. The default n=10 recommendations are requested. The result is a DataFrame stored in user_recommendations.
- print(f"\nTop 10 Recommendations for User {user_id}:"): Prints a header.
- print(user_recommendations[['title', 'genres', 'estimated_rating']]): Prints selected columns ('title', 'genres', 'estimated_rating') from the user_recommendations DataFrame. This displays the list of recommended movies and their predicted scores for the target user.

Hybrid-Based Recommendation System

Content:

- Combines multiple recommendation techniques to overcome the limitations of individual methods.
- Leverages the strengths of both Content-Based and Collaborative Filtering.
- Common combination strategies include:
  - Weighted: Combine scores from different models.
  - Switching: Use one model if possible, switch to another if not.
  - Mixed: Present recommendations from different models side-by-side.
  - Feature Combination: Use features from one model in another.
  - Model-Based: Use one model's output as input for another.
- This script demonstrates a weighted hybrid approach, combining SVD-based predicted ratings (Collaborative) and TF-IDF/Cosine Similarity scores (Content-Based).

Data Loading - Hybrid-Based System

Content:

- Loading the ratings.csv and movies.csv datasets from the specified Google Drive path.
- These datasets are used for both the collaborative (ratings) and content (movie metadata) components of the hybrid system.

First Cell Explanation

```
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(n_factors=100, random_state=42)

svd.fit(trainset)
```

Explanation

- svd = SVD(n_factors=100, random_state=42): Initializes a standard surprise.SVD model instance. n_factors=100 sets the dimensionality of the latent factor vectors to 100. random_state=42 ensures reproducible initialization.
- svd.fit(trainset): Trains the SVD model using the provided trainset. The model learns the user and item latent factors that best approximate the known ratings by minimizing the prediction error through an optimization process (typically stochastic gradient descent) over multiple epochs (using default SVD epoch settings, not explicitly specified here).

Second Cell Explanation

```
tfidf = TfidfVectorizer(stop_words='english',
max_features=10000)
tfidf_matrix =
tfidf.fit_transform(movies['metadata'])
```

Explanation

- tfidf = TfidfVectorizer(stop_words='english', max_features=10000): Initializes a TfidfVectorizer.
- stop_words='english': Removes common English stop words.
- max_features=10000: Limits the vocabulary to the top 10,000 most frequent terms.
- tfidf_matrix = tfidf.fit_transform(movies['metadata']): Fits the vectorizer to the 'metadata' column (learning the vocabulary and IDF values) and transforms the text into a sparse TF-IDF matrix. Each row is a movie, columns are terms, and values are TF-IDF scores. This matrix is the basis for content similarity calculations.

Third Cell Explanation

```
def get_collaborative_recommendations(user_id, model, movies_df, n=10):
    all_movie_ids = movies_df['movieId'].unique()
    rated_movies = ratings[ratings['userId'] == user_id]['movieId'].unique()
    unrated_movies = list(set(all_movie_ids) - set(rated_movies))

    testset = [[user_id, movie_id, 0] for movie_id in unrated_movies]
    predictions = model.test(testset)

    top_predictions = sorted(predictions, key=lambda x: x.est, reverse=True)[:n*2]
    return pd.DataFrame({
        'movieId': [pred.iid for pred in top_predictions],
        'predicted_rating': [pred.est for pred in top_predictions]
    })
```

Explanation

- **def get_collaborative_recommendations(user_id, model, movies_df, n=10)::** Defines a function to get collaborative scores. It takes user_id, the svd_model, the movies_df, and a desired number of *final* recommendations n.
- all_movie_ids = movies_df['movieId'].unique(): Gets all unique movie IDs from the movies_df.
- rated_movies = ratings[ratings['userId'] == user_id]['movieId'].unique(): Finds the movies rated by the specific user from the ratings DataFrame.
- unrated_movies = list(set(all_movie_ids) - set(rated_movies)): Finds the set difference to get movies in the catalog that the user has not rated.
- testset = [[user_id, movie_id, 0] for movie_id in unrated_movies]: Creates a test set for the Surprise model to predict ratings for the unrated movies.
- predictions = model.test(testset): Gets the predicted ratings for the unrated movies.
- top_predictions = sorted(predictions, key=lambda x: x.est, reverse=True)[:n*2]: Sorts predictions and takes the top n*2 results. This provides a larger list for the hybrid function to work with.
- return pd.DataFrame({...}): Returns a pandas DataFrame containing the 'movieId' and the 'predicted_rating' for the top n*2 collaborative recommendations.

Fourth Cell Explanation

```
def get_content_recommendations(movie_title, tfidf_matrix, movies_df, n=10):
    movie_idx = movies_df.index[movies_df['title'] == movie_title].tolist()
    if not movie_idx:
        return pd.DataFrame()
    movie_idx = movie_idx[0]

    cosine_sim = cosine_similarity(tfidf_matrix[movie_idx], tfidf_matrix).flatten()
    top_indices = cosine_sim.argsort()[-n-1:-1][::-1]

    return pd.DataFrame({
        'movieId': movies_df.iloc[top_indices]['movieId'].values,
        'similarity_score': cosine_sim[top_indices]
    })
```

Explanation

- **def get_content_recommendations(movie_title, tfidf_matrix, movies_df, n=10)::** Defines a function to get content-based scores. It takes a movie_title (as the reference), the tfidf_matrix, the movies_df, and the desired number of *final* recommendations n.
- movie_idx = movies_df.index[movies_df['title'] == movie_title].tolist(): Finds the index of the reference movie_title in the movies_df.
- if not movie_idx: return pd.DataFrame(): Returns an empty DataFrame if the movie title is not found.
- movie_idx = movie_idx[0]: Gets the integer index.
- cosine_sim = cosine_similarity(tfidf_matrix[movie_idx], tfidf_matrix).flatten(): Calculates the cosine similarity between the reference movie's TF-IDF vector (tfidf_matrix[movie_idx]) and all other movie vectors in the tfidf_matrix. .flatten() converts the resulting 2D array into a 1D array of scores.
- top_indices = cosine_sim.argsort()[-n-1:-1][::-1]: Finds the indices of the top n*2 similar movies.
- cosine_sim.argsort(): Gets indices sorted by similarity (ascending).
- [-n-1:-1]: Selects the last n*2 indices *before* the very last one (which is the reference movie itself with similarity 1).
- [::-1]: Reverses these selected indices to get them in descending order of similarity.
- return pd.DataFrame({...}): Returns a pandas DataFrame with 'movieId' and 'similarity_score' for the top n*2 content recommendations.

Fifth Cell Explanation

```
def hybrid_recommendations(user_id, movie_title, svd_model, tfidf_matrix, movies_df, n=10):

    collab_recs = get_collaborative_recommendations(user_id, svd_model, movies_df, n*2)
    content_recs = get_content_recommendations(movie_title, tfidf_matrix, movies_df, n*2)

    if collab_recs.empty and content_recs.empty:
        return pd.DataFrame()

    collab_min = collab_recs['predicted_rating'].min() if not collab_recs.empty else 0
    collab_max = collab_recs['predicted_rating'].max() if not collab_recs.empty else 1
    content_min = content_recs['similarity_score'].min() if not content_recs.empty else 0
    content_max = content_recs['similarity_score'].max() if not content_recs.empty else 1

    all_movies = set(collab_recs['movieId']).union(set(content_recs['movieId']))
    hybrid_df = pd.DataFrame({'movieId': list(all_movies)})
```

Fifth Cell Explanation

```
hybrid_df = pd.merge(hybrid_df, collab_recs, on='movieId', how='left')

    hybrid_df = pd.merge(hybrid_df, content_recs, on='movieId', how='left')


    hybrid_df['predicted_rating_norm'] = (hybrid_df['predicted_rating'] - collab_min) / (collab_max - collab_min + 1e-10)

    hybrid_df['similarity_score_norm'] = (hybrid_df['similarity_score'] - content_min) / (content_max - content_min + 1e-10)


    hybrid_df.fillna(0, inplace=True)

    hybrid_df['hybrid_score'] = 0.5 * hybrid_df['predicted_rating_norm'] + 0.5 * hybrid_df['similarity_score_norm']


    top_hybrid = hybrid_df.sort_values('hybrid_score', ascending=False).head(n)

    top_hybrid = pd.merge(top_hybrid, movies_df[['movieId', 'title', 'genres']], on='movieId', how='left')


    return top_hybrid[['movieId', 'title', 'genres', 'hybrid_score', 'predicted_rating', 'similarity_score']]
```

Explanation

- **def hybrid_recommendations(user_id, movie_title, svd_model, tfidf_matrix, movies_df, n=10)::** Defines the main hybrid function. It takes user_id (for collaborative), movie_title (for content), the trained svd_model, the tfidf_matrix, the movies_df, and the desired final number of recommendations n.
- collab_recs = get_collaborative_recommendations(...): Calls the function to get a DataFrame of potential collaborative recommendations (top n*2).
- content_recs = get_content_recommendations(...): Calls the function to get a DataFrame of potential content-based recommendations (top n*2).
- if collab_recs.empty and content_recs.empty: return pd.DataFrame(): Checks if both component recommenders returned empty DataFrames. If so, it returns an empty DataFrame, indicating no recommendations could be generated.
- collab_min = ..., collab_max = ..., content_min = ..., content_max = ...: Calculates the minimum and maximum scores from the collaborative and content recommendation DataFrames. This is necessary for normalizing the scores later. It includes checks (if not df.empty) to handle cases where one of the component recommenders returned an empty list; in such cases, default min/max values (0/1) are used to avoid errors during normalization.

Explanation

- all_movies = set(collab_recs['movieId']).union(set(content_recs['movieId'])): Creates a set containing all unique movie IDs that appear in *either* the collab_recs or content_recs DataFrames. Using a set ensures uniqueness.
- hybrid_df = pd.DataFrame({'movieId': list(all_movies)}): Creates a new DataFrame hybrid_df with a single column 'movieId' containing all these unique movie IDs.
- hybrid_df = pd.merge(hybrid_df, collab_recs, on='movieId', how='left'): Merges hybrid_df with collab_recs on 'movieId'. how='left' ensures all movies in hybrid_df are kept. Movies present only in hybrid_df (i.e., came from content-based but not collab) will have NaN in the 'predicted_rating' column.
- hybrid_df = pd.merge(hybrid_df, content_recs, on='movieId', how='left'): Merges the result with content_recs. Movies present only in hybrid_df (i.e., came from collaborative but not content) will have NaN in the 'similarity_score' column.
- hybrid_df['predicted_rating_norm'] = (...) and hybrid_df['similarity_score_norm'] = (...): These lines perform min-max normalization on the 'predicted_rating' and 'similarity_score' columns. The formula (X−min)/(max−min) scales the values to the range [0, 1]. A small constant 1e-10 is added to the denominator to prevent division by zero if max happens to equal min (e.g., if all scores were the same, though unlikely here). This step is critical for combining scores from different sources effectively.

Explanation

- hybrid_df.fillna(0, inplace=True): Replaces any remaining NaN values in hybrid_df with 0. This handles movies that were only recommended by one of the component systems; their score from the other system is effectively treated as 0 for the hybrid calculation.
- hybrid_df['hybrid_score'] = 0.5 * hybrid_df['predicted_rating_norm'] + 0.5 * hybrid_df['similarity_score_norm']: Calculates the final 'hybrid_score'. It takes a weighted sum of the normalized collaborative and content scores. Here, the weights are 0.5 and 0.5, meaning equal importance is given to both components. These weights are a tunable parameter of the hybrid system.
- top_hybrid = hybrid_df.sort_values('hybrid_score', ascending=False).head(n): Sorts the hybrid_df by the calculated hybrid_score in descending order and selects the top n rows using .head(n). This gives the final list of recommended movies based on the combined score.
- top_hybrid = pd.merge(top_hybrid, movies_df[['movieId', 'title', 'genres']], on='movieId', how='left'): Merges the top_hybrid DataFrame with a subset of the movies_df (containing just 'movieId', 'title', and 'genres') using a left merge on 'movieId'. This adds the movie title and genres to the final recommendation list.
- return top_hybrid[['movieId', 'title', 'genres', 'hybrid_score', 'predicted_rating', 'similarity_score']]: Selects and returns the desired columns for the final output: 'movieId', 'title', 'genres', the combined 'hybrid_score', and the individual 'predicted_rating' (unnormalized) and 'similarity_score' (unnormalized) for context.

Sixth Cell Explanation

```
recommendations = hybrid_recommendations(
    user_id=1,
    movie_title='Iron Man',
    svd_model=svd,
    tfidf_matrix=tfidf_matrix,
    movies_df=movies
)

print(recommendations)
```

Explanation

- recommendations = hybrid_recommendations(...): Calls the hybrid_recommendations function with specified inputs: user_id=1, movie_title='Iron Man' (used by the content component as a seed), the trained svd_model, the tfidf_matrix, and the movies DataFrame. The default n=10 recommendations are requested. The resulting DataFrame is stored in the recommendations variable.
- print(recommendations): Prints the resulting DataFrame of hybrid recommendations. The output will show the recommended movie IDs, titles, genres, the calculated hybrid score, and the original predicted rating (from SVD) and similarity score (from Content-Based) for each recommendation. This demonstrates the output of the hybrid system and how it combines the influence of both components.