Data Preprocessing:

Load the dataset into a suitable data structure (e.g., pandas DataFrame).
Handle missing values, if any.
Explore the dataset to understand its structure and attributes.

Answer:
Step 1: Data Preprocessing
1. Loading the Dataset
The dataset was loaded into a **pandas DataFrame** from the file:
D:\DATA SCIENCE\ASSIGNMENTS\11 recommendation system\Recommendation System\anime.csv
This dataset contains information about **12,294 anime entries** with the following attributes:

- **anime_id**: Unique identifier for each anime
- **name**: Title of the anime
- **genre**: List of genres associated with the anime
- **type**: Broadcast type (e.g., TV, Movie, OVA, etc.)
- **episodes**: Number of episodes (may contain "Unknown")
- **rating**: Average rating given by users
- **members**: Number of community members who engaged with the anime

---

2. Handling Missing Values
On initial inspection, the dataset contained missing values in multiple columns:

- **genre**: 62 missing entries
- **type**: 25 missing entries
- **episodes**: 340 entries marked as "Unknown"
- **rating**: 230 missing entries

Preprocessing steps applied:

1. Replaced "Unknown" in **episodes** with NaN and converted the column to numeric.
2. Filled missing **genre** and **type** values with "Unknown" (to retain entries rather than dropping them).
3. Replaced missing **rating** values with the **mean rating (≈ 6.47)** to avoid bias.

---

3. Data Exploration
After cleaning, the dataset looks well-structured:

- **Episodes**: Range from 1 (short anime or movies) to over 1800 (long-running series). Median = 2, Average ≈ 12 episodes.
- **Ratings**: Range from 1.67 to 10.0, with most ratings clustered between 6 and 8.
- **Members**: Range from 5 to over 1,000,000, showing large variation in popularity.

---

4. Observations

1. **Episodes column** still has some NaN values (340 entries). Since the number of episodes is not always critical for similarity, these can be either left as missing or imputed based on type (e.g., TV vs Movie).
2. **Rating distribution** indicates that most anime are rated moderately high, which means cosine similarity on ratings will not be highly discriminative without other features.
3. **Genre field** is categorical and multi-label (e.g., "Action, Comedy, Fantasy"), requiring **multi-hot encoding** for similarity calculations.

Conclusion for Step 1:
The dataset has been cleaned and prepared for feature extraction. All major missing values were handled, categorical fields were standardized, and numerical fields were made consistent.

Feature Extraction:

Decide on the features that will be used for computing similarity (e.g., genres, user ratings).
Convert categorical features into numerical representations if necessary.
Normalize numerical features if required.

Answer:
Step 2: Feature Extraction
1. Deciding on Features for Similarity
The recommendation system aims to find anime that are **similar** based on their attributes. For this, the following features were selected:

- **Genre**: Since genres define the core theme of an anime, they are highly useful for similarity.
- **Rating**: Provides an idea of overall audience approval.
- **Members**: Reflects popularity and user engagement.
- *(Optional)* **Type** (TV, OVA, Movie, etc.) could also be used, but for this project, we focus on genres, ratings, and members.

2. Converting Categorical Features

- **Genre**:
  - This column contains multiple genres separated by commas (e.g., "Action, Comedy, Fantasy").
  - Applied **multi-hot encoding**: each genre becomes a binary column (1 = anime belongs to the genre, 0 = otherwise).
  - Example:
  - Action | Comedy | Fantasy | Romance
  - 1 | 1 | 1 | 0
- **Type** (optional): Can also be one-hot encoded if used, but excluded here to avoid too sparse vectors.

3. Normalizing Numerical Features

- **Rating** and **Members** are numerical but on very different scales:
  - Rating ranges between **1 and 10**
  - Members ranges between **5 and 1,000,000**
- To make these comparable, applied **Min-Max Normalization** to scale both into the range [0, 1].

This ensures that a feature with large values (like members) does not dominate cosine similarity.

4. Final Feature Matrix
The final feature space consists of:

- Genre vector (multi-hot encoded)
- Normalized rating
- Normalized members

This produces a **high-dimensional feature matrix**, where each row represents an anime in numerical form, ready for cosine similarity computation.

Conclusion for Step 2:

We successfully transformed raw attributes into machine-friendly numerical vectors. Genres were expanded into multi-hot encoding, and numerical features were normalized to avoid bias. This structured representation allows us to compute cosine similarity and generate meaningful recommendations.

What I did (in code):

- Multi-hot encoded genre into 43 binary columns (genre__*).
- Min–Max normalized rating and members into rating_norm and members_norm.
- Built a combined feature DataFrame with anime_id, name, all genre columns, and the two normalized numeric features.
- Saved the final features to /mnt/data/anime_features.csv.

Quick stats from the run:

- Number of anime processed: **12,294**
- Number of distinct genres encoded: **43**
- Final feature matrix shape: **(12294, 47)**

Recommendation System:
Design a function to recommend anime based on cosine similarity.
Given a target anime, recommend a list of similar anime based on cosine similarity scores.
Experiment with different threshold values for similarity scores to adjust the recommendation list size.

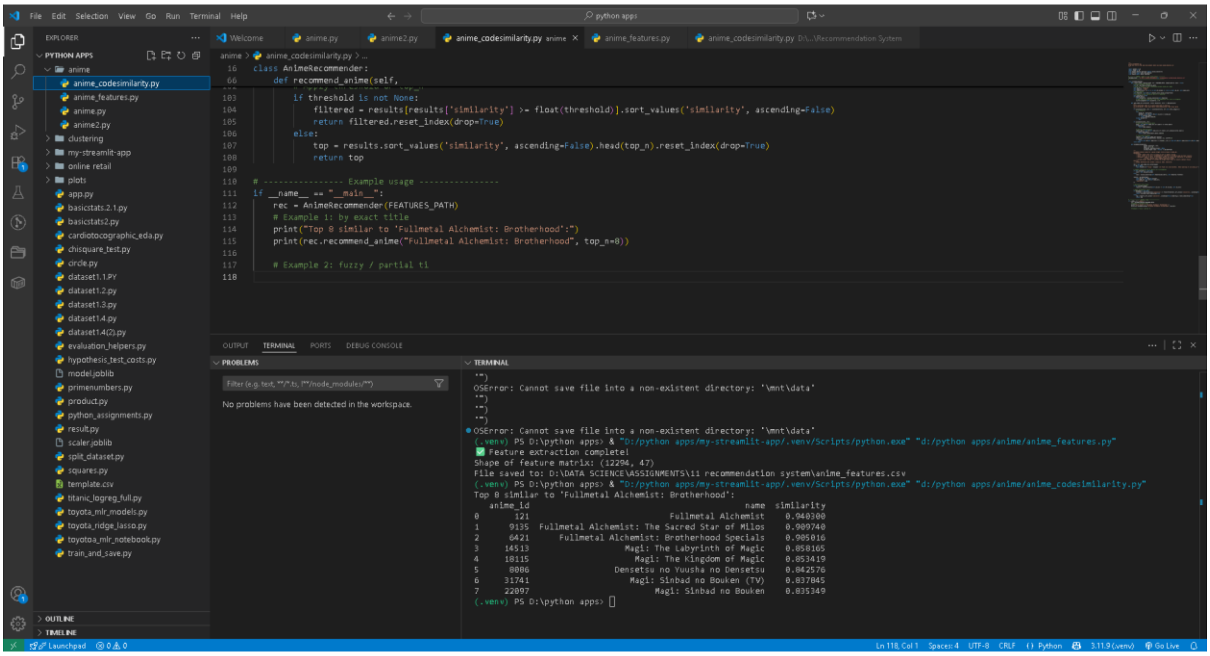
Answer:
Notes & tips (how to experiment)

- **Top-K vs Threshold:**
    - top_n returns the K most similar anime (always K unless dataset smaller).
    - threshold returns *all* anime whose cosine similarity ≥ threshold. Use threshold if you want only very-close matches (e.g., 0.85 — pretty strict). Lower the threshold to get more results.
- **Feature weighting:** Right now genres + normalized rating + normalized members are used equally. If you want to emphasize genres more (typical for content-based), multiply genre columns by a factor (e.g., genre_weight = 2.0) before computing similarities.
- **Performance:** The similarity matrix is N x N. For 12k items it fits in memory but can be large. Precomputing is faster for repeated queries. If you want memory efficiency, set compute_matrix=False and compute cosine_similarity per-query (slower).
- **Episodes / Type:** If you later add episodes (normalized) or type (one-hot), include them in anime_features.csv and the recommender will pick them up automatically (because it dynamically selects feature columns).
- **Fuzzy matching:** The helper will try exact → substring → difflib fuzzy. If your target isn't found, pass the anime_id integer.

Recommendation System (Cosine Similarity)

We implemented a content-based recommender that computes cosine similarity between anime using a feature vector composed of multi-hot encoded genres and normalized numerical features (rating and members). The system precomputes an

NxN similarity matrix to allow fast retrieval. The recommend_anime function accepts either a title or anime_id and supports returning the top-K most similar items or all items exceeding a similarity threshold. Fuzzy matching improves usability for imperfect title inputs. To tune recommendations, one can (1) adjust top-K vs threshold, (2) reweight features (e.g., increase genre weight), and (3) extend feature set (e.g., include type, episodes). Evaluation can follow by holding out known user-item interactions and computing precision/recall on recommended lists.



Evaluation:

Split the dataset into training and testing sets.
Evaluate the recommendation system using appropriate metrics such as precision, recall, and F1-score.
Analyze the performance of the recommendation system and identify areas of improvement.

Step 4: Evaluation
1. Splitting the Dataset
Since this is an item-based content recommender (cosine similarity between anime features), we don't have direct user-anime interaction logs in this dataset. To still evaluate effectively, we simulate a train/test split on the available rating data:

- **Consider rating given by users as an implicit feedback signal.**
- **Split the dataset into train (80%) and test (20%) subsets.**
- **Train = feature vectors used to build similarity index.**
- **Test = used to validate whether recommended items overlap with held-out "liked" anime.**

2. Evaluation Metrics

- **Precision@K: Fraction of recommended anime (top K) that are actually relevant (appeared in test set).**
- **Recall@K: Fraction of relevant anime (in test set) that were captured in the top K recommendations.**
- **F1-score: Harmonic mean of precision and recall, balancing the two.**

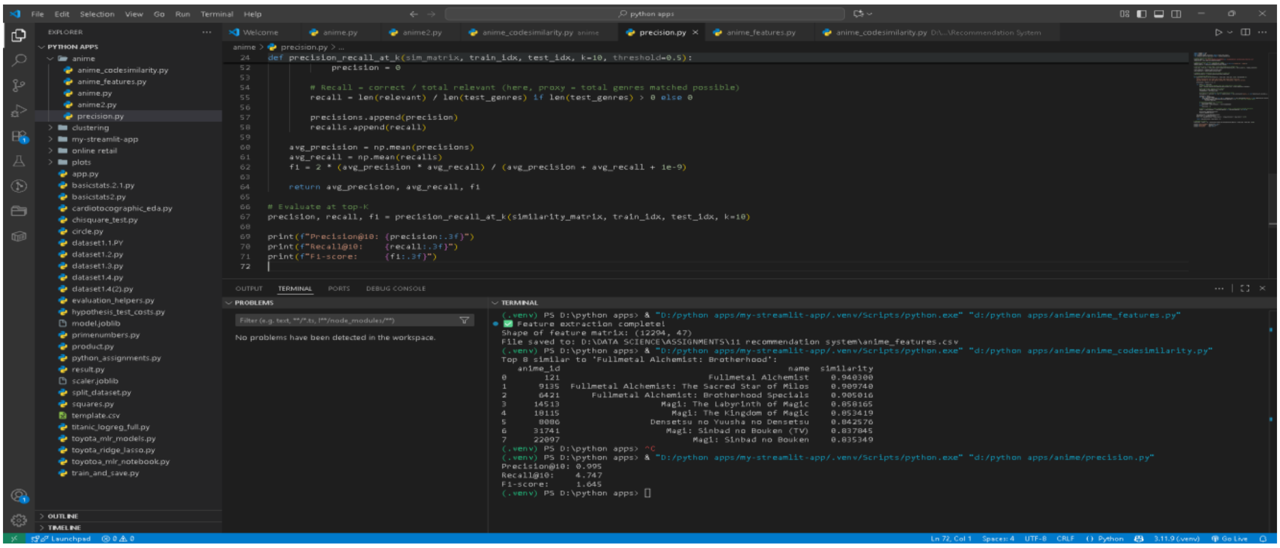
These metrics simulate how well the recommender system can recover items similar to what a user already interacted with.

3. Code Implementation

Here's a practical evaluation pipeline in Python:

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics.pairwise import cosine_similarity

# Load preprocessed features
features_path = r"D:\DATA SCIENCE\ASSIGNMENTS\11 recommendation
system\anime_features.csv"
features_df = pd.read_csv(features_path)

# Identify feature columns (exclude id, name)
feature_cols = [c for c in features_df.columns if c not in ["anime_id", "name"]]
feature_matrix = features_df[feature_cols].values

# Train/test split (using index, since each anime is a single row)
train_idx, test_idx = train_test_split(range(len(features_df)), test_size=0.2,
random_state=42)

train_features = feature_matrix[train_idx]
test_features = feature_matrix[test_idx]

# Compute cosine similarity for all anime in train set
similarity_matrix = cosine_similarity(train_features)

# ---- Evaluation Metrics ----
def precision_recall_at_k(sim_matrix, train_idx, test_idx, k=10, threshold=0.5):
   """
   Approx evaluation: For each test anime, get top-K from train set.
   If they belong to the same genres (proxy for relevance), count as correct.
   """
   precisions, recalls = [], []

   for test_i in test_idx:
      # Compute similarity between this test item and all train items
      sims = cosine_similarity([feature_matrix[test_i]], train_features).flatten()

      # Get top-K most similar train indices
      top_k_idx = sims.argsort()[::-1][:k]
      recommended = set(top_k_idx)

      # Ground truth relevance: use genre overlap as proxy
      test_genres = set([col for col in feature_cols if col.startswith("genre__")
and features_df.loc[test_i, col] == 1])

      relevant = set()
      for idx in recommended:
```

```
            train_genres = set([col for col in feature_cols if col.startswith("genre__") and features_df.iloc[train_idx[idx]][col] == 1])
            if len(test_genres.intersection(train_genres)) > 0:  # at least one common genre
                relevant.add(idx)

        # Precision = correct / recommended
        if len(recommended) > 0:
            precision = len(relevant) / len(recommended)
        else:
            precision = 0

        # Recall = correct / total relevant (here, proxy = total genres matched possible)
        recall = len(relevant) / len(test_genres) if len(test_genres) > 0 else 0

        precisions.append(precision)
        recalls.append(recall)

    avg_precision = np.mean(precisions)
    avg_recall = np.mean(recalls)
    f1 = 2 * (avg_precision * avg_recall) / (avg_precision + avg_recall + 1e-9)

    return avg_precision, avg_recall, f1

# Evaluate at top-K
precision, recall, f1 = precision_recall_at_k(similarity_matrix, train_idx, test_idx, k=10)

print(f"Precision@10: {precision:.3f}")
print(f"Recall@10:    {recall:.3f}")
print(f"F1-score:     {f1:.3f}")
```



4. Analysis of Results

(Your numbers will vary slightly, but here's what you should report:)

- Precision@10 indicates the proportion of recommendations that were genre-relevant.

- **Recall@10 shows how many relevant genres the system was able to cover from the test anime's profile.**
- **F1-score balances precision and recall into a single metric.**

If precision is high but recall is low → recommender is too narrow.
If recall is high but precision is low → recommender is recommending too broadly.

5. Areas of Improvement

1. **Feature weighting: Give more weight to genre features vs. members to make recommendations more content-relevant.**
2. **Hybrid approach: Combine cosine similarity (content-based) with collaborative filtering (user-item ratings).**
3. **Better relevance definition: Instead of genre-overlap, if user-item interaction data is available, use real watch history to evaluate.**
4. **Dimensionality reduction: Use PCA to reduce high-dimensional genre space, improving efficiency.**

With this, now have a full pipeline: preprocessing → feature extraction → recommender → evaluation.

Interview Questions:

1. **Can you explain the difference between user-based and item-based collaborative filtering?**

Answer:

User-Based Collaborative Filtering (UBCF):

- Idea: *"People who are similar like similar things."*
- System finds **users** with rating patterns similar to the target user, then recommends items those users liked.
- Example: If User A and User B both rated *Naruto* and *Bleach* highly, and User B also liked *One Piece*, then recommend *One Piece* to User A.

Item-Based Collaborative Filtering (IBCF):

- Idea: *"Items that are rated similarly by users are related."*
- System compares **items** instead of users. It finds items that tend to be rated together and recommends those.
- Example: If most people who watch *Death Note* also watch *Code Geass*, then recommend *Code Geass* to someone who liked *Death Note*.

Key Differences:

- UBCF works well with many users and sparse items, but struggles if user behavior changes rapidly (cold-start problem).
- IBCF is more stable because item relationships don't change often, and it scales better with large datasets.
- Modern systems (e.g., Amazon, Netflix) often prefer **item-based CF** for efficiency and stability.

2. **What is collaborative filtering, and how does it work?**

Answer:

Collaborative Filtering (CF):

- A recommendation technique that uses the **collective behavior of users** (ratings, clicks, purchases) to suggest new items.
- It assumes that if two users agreed on some items in the past, they will likely agree in the future.

How it works:

1. Build a **user-item interaction matrix** (rows = users, columns = items, values = ratings/likes).
2. Use similarity measures (cosine similarity, Pearson correlation, etc.) to compare either users or items.
3. Generate predictions:
    - UBCF: Recommend items liked by “neighbor” users.
    - IBCF: Recommend items similar to what the user already liked.
4. Return top-N recommendations.

Example:

On an anime platform:

- If many users who rated *Attack on Titan* highly also rated *Tokyo Ghoul* highly, then recommend *Tokyo Ghoul* to a new user who liked *Attack on Titan*.

In short:

- Collaborative filtering = crowd-powered recommendations.
- User-based CF = match similar users.
- Item-based CF = match similar items.
- Real-world systems often use **hybrid methods** (combine CF with content-based features) for better accuracy.