

## 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 ( $\approx 6.47$ )** 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  $\approx 12$  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  $\geq$  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  $\rightarrow$  substring  $\rightarrow$  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.

```

103 def recommend_anime(self,
104     104     105     106     107     108     109     110     111     112     113     114     115     116     117     118
119     120     121     122     123     124     125     126     127     128     129     130     131     132     133     134     135     136     137     138     139     140     141     142     143     144     145     146     147     148     149     150     151     152     153     154     155     156     157     158     159     160     161     162     163     164     165     166     167     168     169     170     171     172     173     174     175     176     177     178     179     180     181     182     183     184     185     186     187     188     189     190     191     192     193     194     195     196     197     198     199     200     201     202     203     204     205     206     207     208     209     210     211     212     213     214     215     216     217     218     219     220     221     222     223     224     225     226     227     228     229     230     231     232     233     234     235     236     237     238     239     240     241     242     243     244     245     246     247     248     249     250     251     252     253     254     255     256     257     258     259     260     261     262     263     264     265     266     267     268     269     270     271     272     273     274     275     276     277     278     279     280     281     282     283     284     285     286     287     288     289     290     291     292     293     294     295     296     297     298     299     300     301     302     303     304     305     306     307     308     309     310     311     312     313     314     315     316     317     318     319     320     321     322     323     324     325     326     327     328     329     330     331     332     333     334     335     336     337     338     339     340     341     342     343     344     345     346     347     348     349     350     351     352     353     354     355     356     357     358     359     360     361     362     363     364     365     366     367     368     369     370     371     372     373     374     375     376     377     378     379     380     381     382     383     384     385     386     387     388     389     390     391     392     393     394     395     396     397     398     399     400     401     402     403     404     405     406     407     408     409     410     411     412     413     414     415     416     417     418     419     420     421     422     423     424     425     426     427     428     429     430     431     432     433     434     435     436     437     438     439     440     441     442     443     444     445     446     447     448     449     450     451     452     453     454     455     456     457     458     459     460     461     462     463     464     465     466     467     468     469     470     471     472     473     474     475     476     477     478     479     480     481     482     483     484     485     486     487     488     489     490     491     492     493     494     495     496     497     498     499     500     501     502     503     504     505     506     507     508     509     510     511     512     513     514     515     516     517     518     519     520     521     522     523     524     525     526     527     528     529     530     531     532     533     534     535     536     537     538     539     540     541     542     543     544     545     546     547     548     549     550     551     552     553     554     555     556     557     558     559     560     561     562     563     564     565     566     567     568     569     570     571     572     573     574     575     576     577     578     579     580     581     582     583     584     585     586     587     588     589     590     591     592     593     594     595     596     597     598     599     600     601     602     603     604     605     606     607     608     609     610     611     612     613     614     615     616     617     618     619     620     621     622     623     624     625     626     627     628     629     630     631     632     633     634     635     636     637     638     639     640     641     642     643     644     645     646     647     648     649     650     651     652     653     654     655     656     657     658     659     660     661     662     663     664     665     666     667     668     669     670     671     672     673     674     675     676     677     678     679     680     681     682     683     684     685     686     687     688     689     690     691     692     693     694     695     696     697     698     699     700     701     702     703     704     705     706     707     708     709     710     711     712     713     714     715     716     717     718     719     720     721     722     723     724     725     726     727     728     729     730     731     732     733     734     735     736     737     738     739     740     741     742     743     744     745     746     747     748     749     750     751     752     753     754     755     756     757     758     759     760     761     762     763     764     765     766     767     768     769     770     771     772     773     774     775     776     777     778     779     780     781     782     783     784     785     786     787     788     789     790     791     792     793     794     795     796     797     798     799     800     801     802     803     804     805     806     807     808     809     810     811     812     813     814     815     816     817     818     819     820     821     822     823     824     825     826     827     828     829     830     831     832     833     834     835     836     837     838     839     840     841     842     843     844     845     846     847     848     849     850     851     852     853     854     855     856     857     858     859     860     861     862     863     864     865     866     867     868     869     870     871     872     873     874     875     876     877     878     879     880     881     882     883     884     885     886     887     888     889     890     891     892     893     894     895     896     897     898     899     900     901     902     903     904     905     906     907     908     909     910     911     912     913     914     915     916     917     918     919     920     921     922     923     924     925     926     927     928     929     930     931     932     933     934     935     936     937     938     939     940     941     942     943     944     945     946     947     948     949     950     951     952     953     954     955     956     957     958     959     960     961     962     963     964     965     966     967     968     969     970     971     972     973     974     975     976     977     978     979     980     981     982     983     984     985     986     987     988     989     990     991     992     993     994     995     996     997     998     999     1000
111 if __name__ == "__main__":
112     rec = AnimeRecommender(FEATURES_PATH)
113     # Example 1: by exact title
114     print("Top 8 similar to 'Fullmetal Alchemist: Brotherhood':")
115     print(rec.recommend_anime("Fullmetal Alchemist: Brotherhood", top_n=8))
116
117     # Example 2: Fuzzy / partial ti
118

```

```

PS D:\python_apps> python.exe "d:/python_apps/anime/anime_codesimilarity.py"
Feature extraction complete!
File saved to: D:\DATA\SCIENCE\ASSIGNMENTS\all_recommendation_system\anime_features.csv
PS D:\python_apps> python.exe "d:/python_apps/anime/anime_codesimilarity.py"
Top 8 similar to 'Fullmetal Alchemist: Brotherhood':
anime_id      name      similarity
0      121      Fullmetal Alchemist      0.949189
1      9135      Fullmetal Alchemist: The Sacred Star of Milos      0.909740
2      6421      Fullmetal Alchemist: Brotherhood Specials      0.909316
3      14513      Magi: The Labyrinth of Magic      0.893105
4      18115      Magi: The Kingdom of Magic      0.893419
5      40880      Densetsu no Yūsha no Densetsu      0.842576
6      31741      Magi: Sinbad no Bouken (TV)      0.877885
7      22897      Magi: Sinbad no Bouken      0.835349

```

## 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}")

```

The screenshot shows a Jupyter Notebook with a file explorer on the left, a code editor in the center, and a terminal/output area at the bottom. The code in the notebook is the same as the one provided in the previous blocks. The terminal output shows the results of the evaluation:

```

[venv] PS D:\python apps> "D:\python apps\my-streamlit-app\venv\scripts\python.exe" "D:\python apps\anime\anime_features.py"
Feature extraction completed
Shape of feature matrix: (12294, 47)
File saved to: D:\DATA SCIENCE\ASSIGNMENT5\11 recommendation system\anime_features.csv
[venv] PS D:\python apps> "D:\python apps\my-streamlit-app\venv\scripts\python.exe" "D:\python apps\anime\anime_codesimilarity.py"
Top 8 similar to 'Fullmetal Alchemist: Brotherhood':
anime_id    name    similarity
0    121    Fullmetal Alchemist    0.96308
1    4128    Fullmetal Alchemist: The Sacred Star of Milos    0.899748
2    6421    Fullmetal Alchemist: Brotherhood Specials    0.906016
3    14519    Magi: The Labyrinth of Magic    0.858165
4    18135    Magi: The Kingdom of Magic    0.853419
5    8880    Denzetsu no Yousha no Denzetsu    0.842576
6    31741    Magi: Sinbad no Bouken (TV)    0.837845
7    22697    Magi: Sinbad no Bouken    0.835349
[venv] PS D:\python apps> "D:\python apps\my-streamlit-app\venv\scripts\python.exe" "D:\python apps\anime\precision.py"
Precision@10: 0.905
Recall@10: 4.747
F1-score: 1.045
[venv] PS D:\python apps>

```

## 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.