Name: Subhapreet Patro

Roll No.: 2211CS010547

Group: 3

Dataset Link: https://www.kaggle.com/datasets/tamber/steam-video-games/data

In [4]:
```python
import pandas as pd
```

In [5]:
```python
df=pd.read_csv("steam-200k.csv")
df
```

Out[5]:

	UserID	Game	Action	Hours	Flag
0	151603712	The Elder Scrolls V Skyrim	purchase	1.0	0
1	151603712	The Elder Scrolls V Skyrim	play	273.0	0
2	151603712	Fallout 4	purchase	1.0	0
3	151603712	Fallout 4	play	87.0	0
4	151603712	Spore	purchase	1.0	0
...	...	...	...	...	...
199995	128470551	Titan Souls	play	1.5	0
199996	128470551	Grand Theft Auto Vice City	purchase	1.0	0
199997	128470551	Grand Theft Auto Vice City	play	1.5	0
199998	128470551	RUSH	purchase	1.0	0
199999	128470551	RUSH	play	1.4	0

200000 rows × 5 columns

Steam Video Games Dataset (200K Records)

Dataset Overview

This dataset contains user interactions with video games on Steam, including purchases and playtime data. It consists of 200,000 records, covering thousands of users and games. The dataset can be used for building recommendation systems, analyzing user behavior, and understanding gaming trends.

Files

- `steam-200k.csv` : The main dataset containing user interactions with games.

Columns Description

Column	Data Type	Description
`UserID`	`int64`	Unique identifier for each Steam user.
`Game`	`object`	Name of the game associated with the user action.
`Action`	`object`	Indicates whether the user **purchased** the game (`"purchase"`) or **played** the game (`"play"`).
`Hours`	`float64`	The number of hours the user has played the game. For `"purchase"` actions, this value is always `1.0` .
`Flag`	`int64`	This column is always `0` and does not contain useful information.

Data Summary

- **Total Records**: 200,000
- **Unique Users**: ~123,000
- **Unique Games**: ~5,000
- **Actions**: `"purchase"` or `"play"`
- **Playtime Range**: 0.1 to 11,754 hours
- **Median Playtime**: 1.0 hour
- **Most Users Play Less Than**: 1.3 hours (75th percentile)

Key Insights & Observations

1. User Behavior Trends

- Users can have multiple entries in the dataset for different games.
- Some users only have `"purchase"` records, while others have `"play"` records.
- There are extreme outliers where some users have played over `10,000` hours.

2. Game Popularity Analysis

- The dataset can help identify the most purchased vs. most played games.
- Certain games may have high playtime but low purchase frequency, which could indicate **free-to-play** games.

3. Player Engagement Patterns

- Most players have low playtime (`< 2 hours`), suggesting many games are either **casual** or not engaging enough.
- A few hardcore users play certain games extensively, crossing `1,000+` hours.

In [7]:
```
df.shape
```

Out[7]:
```
(200000, 5)
```

`df.shape`

The `df.shape` function in Pandas returns the dimensions of the dataset as a tuple `(rows, columns)` .

In [9]:
```
df.isnull().sum()
```

Out[9]:
```
UserID    0
Game      0
Action    0
Hours     0
Flag      0
dtype: int64
```

`df.isnull().sum()`

The `df.isnull().sum()` function in Pandas is used to check for missing (null) values in each column of the dataset. It returns a count of `NaN` (Not a Number) values for every column.

In [11]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 200000 entries, 0 to 199999
Data columns (total 5 columns):
 #   Column  Non-Null Count   Dtype  
---  ------  --------------   -----  
 0   UserID  200000 non-null  int64  
 1   Game    200000 non-null  object 
 2   Action  200000 non-null  object 
 3   Hours   200000 non-null  float64
 4   Flag    200000 non-null  int64  
dtypes: float64(1), int64(2), object(2)
memory usage: 7.6+ MB
```

df.info()

The `df.info()` function in Pandas provides a concise summary of the dataset, including the **number of entries, column names, data types, and memory usage**.

In [13]:
```
from sklearn.metrics.pairwise import cosine_similarity
from scipy.sparse import csr_matrix

df = pd.read_csv("steam-200k.csv")

df_play = df[df["Action"] == "play"].drop(columns=["Action", "Flag"])

ratings = df_play.pivot_table(index="UserID", columns="Game", values="Hours", fill_value=0)

ratings_sparse_matrix = csr_matrix(ratings.values)

item_similarity = cosine_similarity(ratings_sparse_matrix.T)
similarity_df = pd.DataFrame(item_similarity, index=ratings.columns, columns=ratings.columns)

def collaborative_filtering(user, ratings, similarity_df):
    if user not in ratings.index:
        return "User not found in dataset."

    user_ratings = ratings.loc[user]
    scores = {}

    for game in ratings.columns:
        if user_ratings[game] == 0:
            sim_games = similarity_df[game]
            rated_games = user_ratings[user_ratings > 0].index
            score = sum(sim_games[rated_game] * user_ratings[rated_game] for rated_game in rated_games)
            scores[game] = score

    return sorted(scores.items(), key=lambda x: x[1], reverse=True)

sample_user = ratings.index[547]
recommendations = collaborative_filtering(sample_user, ratings, similarity_df)

print(f"Top 5 recommendations for User {sample_user}:")
for game, score in recommendations[:5]:
    print(f"{game}: {score:.4f}")
```

```
Top 5 recommendations for User 30548861:
David.: 0.0487
Knightmare Tower: 0.0487
Violett: 0.0487
Larva Mortus: 0.0292
Infestation Survivor Stories: 0.0275
```

Code Explanation

This code demonstrates a collaborative filtering approach to recommend games for a user based on their play history using cosine similarity. Here's a breakdown of the main components:

1. **Loading the Data:** The dataset `steam-200k.csv` is loaded into a Pandas DataFrame. The dataset contains user interactions with games, including information like the `UserID` , `Game` , `Action` (such as "play"), and `Hours` spent playing.
2. **Filtering the Data:** The DataFrame is filtered to only include rows where the action is "play". The columns `Action` and `Flag` are then dropped because they are not needed for the collaborative filtering process.
3. **Creating the Ratings Matrix:** A pivot table is created with `UserID` as the index, `Game` as the columns, and `Hours` as the values. Missing values are filled with `0` to indicate that the user has not played that game. This matrix helps in structuring the data such that it's easier to compute similarities between games.
4. **Converting to Sparse Matrix:** The ratings matrix is converted into a sparse matrix format using `csr_matrix` . This format is more efficient for handling large datasets with a lot of zeros (indicating that a user hasn't played a particular game).
5. **Calculating Item Similarity:** Cosine similarity is calculated between the games (items) using the sparse ratings matrix. The result is a similarity matrix that shows how similar each game is to every other game. This similarity matrix is stored in a DataFrame called `similarity_df` .
6. **Collaborative Filtering Function:** The `collaborative_filtering` function takes a user, the ratings matrix, and the similarity matrix as inputs. It checks for games that the user hasn't played yet and calculates a score for each of these games. The score is based on the similarity of the games the user has already played, weighted by the number of hours spent on those games.
7. **Generating Recommendations:** For a given `sample_user` , the `collaborative_filtering` function generates recommendations based on games that are similar to the ones the user has played. The function looks for the user's ratings and calculates the score for games they haven't played yet.
8. **Displaying the Top 5 Recommendations:** The top 5 recommended games are displayed, sorted by their calculated scores in descending order. These games are the most likely suggestions for the user based on the similarity to the games they have already played.

In [15]:
```python
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.feature_extraction.text import TfidfVectorizer

df = pd.read_csv("steam-200k.csv")

df_play = df[df["Action"] == "play"].drop(columns=["Action", "Flag"])

metadata = df_play.groupby("Game")["Hours"].mean().reset_index()
metadata["Features"] = metadata["Game"]

tfidf = TfidfVectorizer(stop_words='english')
feature_matrix = tfidf.fit_transform(metadata["Features"])

content_similarity = cosine_similarity(feature_matrix)
content_similarity_df = pd.DataFrame(content_similarity, index=metadata["Game"], columns=metadata["Game"])

ratings = df_play.pivot_table(index="UserID", columns="Game", values="Hours", fill_value=0)

def content_based_filtering(user, ratings, content_similarity_df):
    if user not in ratings.index:
        return "User not found in dataset."

    user_ratings = ratings.loc[user]
    scores = {}
    for game in ratings.columns:
        if user_ratings[game] == 0:
            sim_games = content_similarity_df[game]
            rated_games = user_ratings[user_ratings > 0].index
            score = sum(sim_games[rated_game] * user_ratings[rated_game] for rated_game in rated_games)
            scores[game] = score

    return sorted(scores.items(), key=lambda x: x[1], reverse=True)

sample_user = ratings.index[619]
recommendations = content_based_filtering(sample_user, ratings, content_similarity_df)

print(f"Content-Based Filtering Recommendations for UserID: {sample_user}:")
for game, score in recommendations[:5]:
    print(f"{game}: {score:.4f}")
```

```
Content-Based Filtering Recommendations for UserID: 33457161:
Counter-Strike Source: 60.7958
Counter-Strike Nexon Zombies: 50.4672
Counter-Strike Global Offensive: 50.0858
Counter-Strike Condition Zero Deleted Scenes: 40.3517
Fair Strike: 31.3531
```

Code Explanation

This code demonstrates a content-based filtering approach to recommend games for a user based on the features of the games themselves, rather than relying on other users' ratings. It uses cosine similarity to compare the content of the games. Here's an explanation of the key parts of the code:

1. **Loading the Data:** The dataset `steam-200k.csv` is loaded into a Pandas DataFrame. This dataset contains user interactions with games, including `UserID`, `Game`, `Action` (such as "play"), and `Hours` spent playing.
2. **Filtering the Data:** The DataFrame is filtered to include only the rows where the action is "play". The `Action` and `Flag` columns are dropped because they are not necessary for the content-based filtering process.
3. **Creating Metadata for Games:** The `metadata` DataFrame is created by grouping the data by `Game` and calculating the average `Hours` spent playing each game. This provides a summary of how long users typically play each game.
4. **Feature Extraction:** A new column, `Features`, is added to the `metadata` DataFrame. In this case, the game name itself is used as the feature. This is a simplified approach; ideally, this column would contain more detailed information about the game (such as genre, description, etc.). The `TfidfVectorizer` from `sklearn` is used to transform the game names into a matrix of numerical features. The `stop_words='english'` argument removes common English stop words from the game names during the vectorization process. This creates a feature matrix that represents the textual content of the games.
5. **Calculating Content Similarity:** The `cosine_similarity` function is used to calculate the similarity between the game features. This results in a content similarity matrix, `content_similarity_df`, that measures how similar each game is to every other game based on their textual features.
6. **Creating the Ratings Matrix:** A pivot table is created with `UserID` as the index, `Game` as the columns, and `Hours` as the values. Missing values (where a user hasn't played a game) are filled with `0`.
7. **Content-Based Filtering Function:** The `content_based_filtering` function takes a user, the ratings matrix, and the content similarity matrix as inputs. It checks for games that the user hasn’t played yet and calculates a score for each of these games. The score is based on the similarity of the game to the ones the user has already played, weighted by the number of hours the user has spent playing those games.
8. **Generating Recommendations:** For a given `sample_user`, the `content_based_filtering` function generates recommendations based on games that are similar to the ones the user has played. The function calculates a score for each unplayed game based on the similarity to the games the user has rated.
9. **Displaying the Top 5 Recommendations:** The top 5 recommended games are displayed, sorted by their calculated scores in descending order. These games are the most likely suggestions for the user based on the similarity of the content of the games they have already played.

In [17]:
```python
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.feature_extraction.text import TfidfVectorizer
from scipy.sparse import csr_matrix

df = pd.read_csv("steam-200k.csv")

df_play = df[df["Action"] == "play"].drop(columns=["Action", "Flag"])

ratings = df_play.pivot_table(index="UserID", columns="Game", values="Hours", fill_value=0)

ratings_sparse_matrix = csr_matrix(ratings.values)

item_similarity = cosine_similarity(ratings_sparse_matrix.T)
similarity_df = pd.DataFrame(item_similarity, index=ratings.columns, columns=ratings.columns)

vectorizer = TfidfVectorizer()
game_tfidf_matrix = vectorizer.fit_transform(ratings.columns)
content_similarity = cosine_similarity(game_tfidf_matrix)
content_similarity_df = pd.DataFrame(content_similarity, index=ratings.columns, columns=ratings.columns)

def collaborative_filtering(user, ratings, similarity_df):
    if user not in ratings.index:
        return []
    user_ratings = ratings.loc[user]
    scores = {}
    for game in ratings.columns:
        if user_ratings[game] == 0:
            sim_games = similarity_df[game]
            rated_games = user_ratings[user_ratings > 0].index
            scores[game] = sum(sim_games[rated_game] * user_ratings[rated_game] for rated_game in rated_games)
    return sorted(scores.items(), key=lambda x: x[1], reverse=True)

def content_based_filtering(user, ratings, content_similarity_df):
    if user not in ratings.index:
        return []
    user_ratings = ratings.loc[user]
    scores = {}
    for game in ratings.columns:
        if user_ratings[game] == 0:
            sim_games = content_similarity_df[game]
            rated_games = user_ratings[user_ratings > 0].index
            scores[game] = sum(sim_games[rated_game] * user_ratings[rated_game] for rated_game in rated_games)
    return sorted(scores.items(), key=lambda x: x[1], reverse=True)

def hybrid_recommendation(user, ratings, similarity_df, content_similarity_df, alpha=0.5):
    collaborative_scores = dict(collaborative_filtering(user, ratings, similarity_df))
```

```
    content_scores = dict(content_based_filtering(user, ratings, content_similarity_df))
    hybrid_scores = {}
    for game in ratings.columns:
        hybrid_scores[game] = alpha * collaborative_scores.get(game, 0) + (1 - alpha) * content_scores.get(game, 0)
    return sorted(hybrid_scores.items(), key=lambda x: x[1], reverse=True)

sample_user = ratings.index[297]
recommendations = hybrid_recommendation(sample_user, ratings, similarity_df, content_similarity_df)

print(f"Hybrid Recommendations for UserID: {sample_user}:")
for game, score in recommendations[:5]:
    print(f"{game}: {score:.4f}")
```

```
Hybrid Recommendations for UserID: 18888504:
Counter-Strike: 15.0916
Counter-Strike Global Offensive: 14.5607
Counter-Strike Condition Zero: 10.6859
Counter-Strike Nexon Zombies: 10.0121
Counter-Strike Condition Zero Deleted Scenes: 8.5396
```

Code Explanation

This code implements a hybrid recommendation system that combines both collaborative filtering and content-based filtering approaches. It uses cosine similarity to compare both user-item interactions and the content features of the items (games). Here's a detailed explanation of the key parts of the code:

1. **Loading the Data:** The dataset `steam-200k.csv` is loaded into a Pandas DataFrame. This dataset contains user interactions with games, including `UserID` , `Game` , `Action` (such as "play"), and `Hours` spent playing.
2. **Filtering the Data:** The DataFrame is filtered to only include rows where the action is "play". The `Action` and `Flag` columns are dropped as they are not necessary for the recommendation process.
3. **Creating the Ratings Matrix:** A pivot table is created with `UserID` as the index, `Game` as the columns, and `Hours` as the values. Missing values (where a user hasn't played a particular game) are filled with `0` .
4. **Converting the Ratings Matrix to a Sparse Matrix:** The ratings matrix is converted to a sparse matrix format using `csr_matrix` . This helps efficiently handle large matrices with many zeros (indicating that a user has not rated a particular game).
5. **Calculating Item Similarity (Collaborative Filtering):** The `cosine_similarity` function calculates the similarity between the games based on the ratings matrix. This results in a similarity matrix, `similarity_df` , that measures how similar each game is to every other game based on user interactions.
6. **Creating the Game Content Matrix (Content-Based Filtering):** A `TfidfVectorizer` is used to create a TF-IDF matrix for the game names (the features). The resulting `game_tfidf_matrix` represents the textual features of the games, and `cosine_similarity` is then applied to compute the content similarity between games. This results in a content similarity matrix, `content_similarity_df` .
7. **Collaborative Filtering Function:** The `collaborative_filtering` function takes a user, the ratings matrix, and the item similarity matrix as input. It generates a list of recommended games for the user based on how similar the games they have already played are to other unplayed games.

8. **Content-Based Filtering Function:** The `content_based_filtering` function takes a user, the ratings matrix, and the content similarity matrix as input. It generates a list of recommended games for the user based on the similarity of the games' features (such as game names) to those that the user has already played.
9. **Hybrid Recommendation Function:** The `hybrid_recommendation` function combines the collaborative filtering and content-based filtering scores. It takes a `user` , the ratings matrix, the item similarity matrix, and the content similarity matrix as inputs. The `alpha` parameter controls the balance between the collaborative and content-based scores (with `alpha=0.5` giving equal weight to both). The function generates hybrid scores for each game by combining the scores from both recommendation approaches.
10. **Generating Recommendations:** For a given `sample_user` , the `hybrid_recommendation` function generates recommendations by combining the collaborative filtering and content-based filtering scores. The recommended games are sorted in descending order of their combined score.
11. **Displaying the Top 5 Recommendations:** The top 5 recommended games are displayed, sorted by their hybrid scores. These games are suggested to the user based on their past ratings and the content similarities of the games they have played