Batch: E2	**Roll No.: 16010123325**
Experiment No. 7	

Title: To perform Extract, Transform, Load (ETL) using Python

Aim : Load a dataset from external sources (web scraping), apply transformations such as scaling, normalization, and feature encoding, save the transformed data into a structured format.

Course Outcome:

CO**3 :** Learn data cleaning, transformation, and feature engineering techniques.

Books/ Journals/ Websites referred:

1. The Comprehensive R Archive Network
2. Posit

Resources used:

(Students should write the data sources used)

Theory:

What is ETL?

ETL stands for "Extract, Transform, and Load" and describes the set of processes to extract data from one system, transform it, and load it into a target repository. An ETL pipeline is a traditional type of data pipeline for cleaning, enriching, and transforming data from a variety of sources before integrating it for use in data analytics, business intelligence and data science.

Key Benefits

Using an ETL pipeline to transform raw data to match the target system, allows for systematic and accurate data analysis to take place in the target repository. Specifically, the key benefits are:

1. More stable and faster data analysis on a single, pre-defined use case. This is because the data set has already been structured and transformed.
2. Easier compliance with GDPR, HIPAA, and CCPA standards. This is because users can omit any sensitive data prior to loading in the target system.
3. Identify and capture changes made to a database via the change data capture (CDC) process or technology. These changes can then be applied to another data repository or made available in a format consumable by ETL, EAI, or other types of data integration tools.

Extract > Transform > Load (ETL)

In the ETL process, transformation is performed in a staging area outside of the data warehouse and before loading it into the data warehouse. The entire data set must be transformed before loading, so transforming large data sets can take a lot of time up front. The benefit is that analysis can take place immediately once the data is loaded. This is why this process is appropriate for small data sets which require complex transformations.

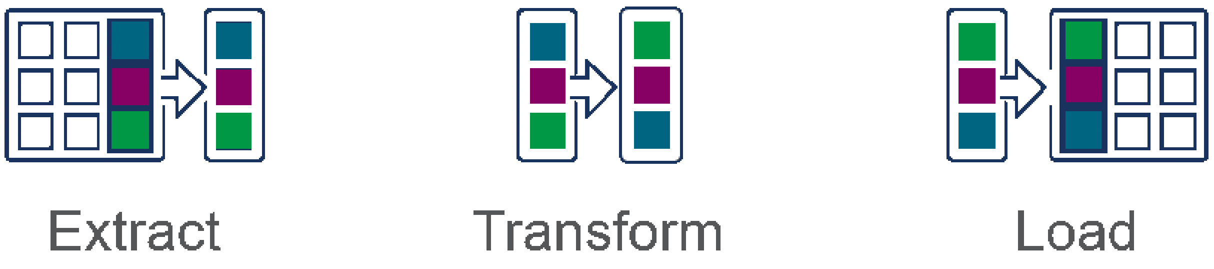


Fig. 1: Illustration showing the 3 steps of the ETL process which are extract, transform and load.

Extract > Load > Transform (ELT)

In the ELT process, data transformation is performed on an as-needed basis within the target system. This means that the ELT process takes less time. But if there is not sufficient processing power in the cloud solution, transformation can slow down the querying and analysis processes. This is why the ELT process is more appropriate for larger, structured and unstructured data sets and when timeliness is important.

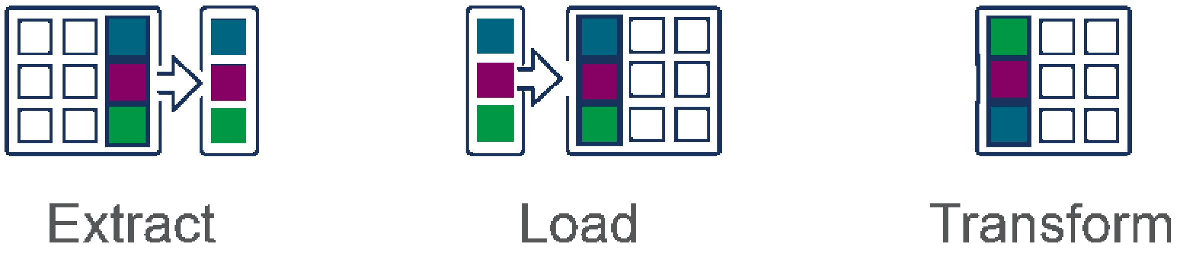


Fig. 2: Illustration showing the 3 steps of the ELT process which are extract, load and transform.

Extract phase

The Extract phase is the first step of the ETL process. It involves gathering raw data from a source. In this case, we collect data from IMDb's Top Rated Movies list using web scraping in Python. Below is a detailed step-by-step explanation of how this is done.

Step 1: Sending an HTTP Request

To access the IMDb page, we send an HTTP request using the requests library.

An HTTP request is like knocking on IMDb's door and asking for a copy of the webpage.

IMDb responds by sending back the HTML code of the page, which contains all the movie details.

Step 2: Parsing the HTML Content

The HTML code received from IMDb is not easy to read directly.

We use the BeautifulSoup library to convert the messy HTML into a structured format that we can work with.

This makes it easier to locate and extract specific movie details like titles, release years, and ratings.

Step 3: Extracting Movie Details

Once the HTML is structured, we search for specific HTML tags that contain the movie information.

We extract the following details for each movie:

Title – The name of the movie

Year – The year the movie was released

Duration – The length of the movie in hours and minutes

IMDb Rating – The rating given by IMDb users

Vote Count – The number of users who rated the movie

We also clean the extracted data by removing unwanted characters.

Step 4: Storing the Extracted Data

After extracting the data, we store it in a Pandas DataFrame (a table format in Python).

This structured table makes it easy to analyze and manipulate the data.

Finally, we save the extracted data as a CSV file so that it can be used later in the next steps of the ETL process.

Transform Phase

The Transform phase is the second step in the ETL process. After extracting raw movie data from IMDb, we need to clean, organize, and modify it so that it becomes useful for analysis. Below is a detailed step-by-step explanation of how this is done.

Step 1: Extracting the Correct Year

Sometimes, the year column may contain extra symbols or unwanted text.

We use Python to extract only the four-digit year (e.g., 1994, 2001) from the raw data.

Step 2: Converting Duration to Minutes

The duration of movies is usually written in the format "2h 30m" (2 hours 30 minutes).

We convert this into just minutes (e.g., "2h 30m" → 150 minutes) so it is easier to analyze.

Step 3: Cleaning Vote Counts

IMDb sometimes shows votes in short forms like "2M" (2 million) or "500K" (500,000).

We convert these values into actual numbers (e.g., "2M" → 2,000,000).

Step 4: Categorizing Movies by Decade

We group movies based on their release decade (e.g., movies from 1990-1999 are in the “1990s” group).

This helps in analyzing trends over different time periods.

Step 5: Classifying Movies by Duration

We divide movies into categories based on their length:

Short (<90 min)

Medium (90-120 min)

Long (120-150 min)

Very Long (150-180 min)

Epic (>180 min)

Step 6: Categorizing Movies by Rating

We group movies based on their IMDb rating:

Low (<7.5)

Average (7.5-8.0)

Good (8.0-8.5)

Excellent (8.5-9.0)

Masterpiece (9.0+)

Step 7: Handling Missing Values

If any movie is missing important details like title, year, or rating, we remove it from the dataset.

Step 8: Removing Duplicates

If the same movie appears multiple times, we remove duplicates to keep the dataset clean.

Step 9: Creating a Popularity Score

A custom popularity score is calculated based on IMDb rating and number of votes.

This helps in ranking movies based on both their quality and popularity.

Load Phase

Once the transformed movie data is ready, it needs to be stored in a structured format for further analysis. This step is called the Load Phase, where we save the processed data into a SQLite database and export it into useful formats.

1. Establish a Connection to the SQLite Database

To store the processed data, we first need to create a SQLite database engine. SQLite is a lightweight database that stores data in a single file, making it perfect for small to medium-sized datasets.

Using the SQLAlchemy create_engine function, we create a database connection:

```
engine = create_engine('sqlite:///data/processed/movies_database.sqlite')
```

This creates (or opens, if it already exists) a database file named movies_database.sqlite inside the data/processed folder.

2. Load the Transformed Data into the Main Table (movies)

The main dataset, stored in a Pandas DataFrame (df), contains individual movie details like title, rating, duration, votes, and popularity. We load this data into a table named movies inside the SQLite database:

```
df.to_sql('movies', engine, if_exists='replace', index=False)
```

This creates the movies table. If it already exists, it replaces it to ensure we always have the latest data.

Excludes the default Pandas index from being stored.

3. Aggregate and Store Data by Decades (decade_stats Table)

To analyze movie trends over time, we group movies by decade and compute:

- Total number of movies per decade
- Average rating
- Average duration
- Average popularity score

```
decade_stats = df.groupby('decade').agg({

    'title': 'count',

    'rating': 'mean',

    'duration_minutes': 'mean',

    'popularity_score': 'mean'

}).reset_index()

decade_stats.to_sql('decade_stats', engine, if_exists='replace', index=False)
```

This creates a new table decade_stats in the database.

4. Aggregate and Store Data by Rating Categories (rating_stats Table)

Another useful breakdown is by rating category (e.g., “Excellent”, “Good”, “Average”). We compute:

- Total movies per category
- Average release year
- Average duration
- Average votes count

```
rating_stats = df.groupby('rating_category').agg({

    'title': 'count',

    'year': 'mean',

    'duration_minutes': 'mean',

    'votes_count': 'mean'

}).reset_index()

rating_stats.to_sql('rating_stats', engine, if_exists='replace', index=False)
```

This creates a new table rating_stats in the database.

5. Export Aggregated Data to CSV Files

To make the aggregated tables available for external use, we save them as CSV files:

```
decade_stats.to_csv('data/processed/decade_stats.csv', index=False)

rating_stats.to_csv('data/processed/rating_stats.csv', index=False)
```

These CSV files allow us to analyze the data in Excel, Pandas, or other tools without requiring database access.

6. Verify Data Integrity with a Record Count

To ensure the data was successfully stored, we run a query to count the number of movies in the database:

```
conn = sqlite3.connect('data/processed/movies_database.sqlite')

movie_count = pd.read_sql("SELECT COUNT(*) FROM movies", conn).iloc[0, 0]

conn.close()
```

This queries the movies table and retrieves the total record count.

Finally, we print the count to confirm the data was successfully loaded.

Visualization and ETL Pipeline

Once data is loaded into the database, visualizing it helps uncover patterns and trends. We create four key visualizations:

1. **Ratings Distribution:**
    - A histogram is used to show how movie ratings are distributed.
    - Helps identify whether most movies have high or low ratings, and if the distribution is skewed.
2. **Movies by Decade:**
    - A bar chart shows how many movies were produced in each decade.
    - Helps analyze how the film industry has grown or declined over time.

3. **Rating vs. Duration:**
    - A scatter plot is used to observe the relationship between movie ratings and duration.
    - Helps determine whether longer movies tend to have higher ratings.
4. **Average Movie Duration by Decade:**
    - A line chart tracks how the average movie length has changed over decades.
    - Useful for analyzing trends in storytelling preferences over time.

These visualizations help summarize large amounts of data into **easy-to-understand insights**, aiding in decision-making and further analysis.

An **ETL (Extract, Transform, Load) pipeline** automates data collection, processing, and storage. Instead of manually handling raw data, we use a structured pipeline to ensure consistency, accuracy, and efficiency.

1. **Extract (E)**
    - Fetches movie data from an external source (e.g., web scraping).
    - Automates data collection, reducing manual effort.
2. **Transform (T)**
    - Cleans and processes the data (e.g., removing duplicates, categorizing movies by decade).
    - Ensures data is structured and meaningful for analysis.
3. **Load (L)**
    - Stores the transformed data in a **SQLite database**.
    - Creates summary tables for easy access and saves results in CSV files for external use.

This pipeline ensures that **data is collected, structured, and stored automatically**, making it easier to analyze trends over time without manual intervention.

Students' task :

Based on the above steps, **Choose a Data Source:**

- Select a website or API that provides structured data (e.g. sports statistics, financial data, weather data, RottenTomatoes, Wikipedia etc.)
    - Ensure the data is accessible and contains meaningful information for analysis.

Perform the ETL Process:

- **Extract:** Write a script to scrape or retrieve data from the selected source.
- **Transform:** Clean the data, categorize it, and make it suitable for analysis.
- **Load:** Store the processed data in a **SQLite database**.

```
!pip install requests pandas matplotlib seaborn sqlalchemy



import requests

import pandas as pd

import numpy as np

import time

import sqlite3

from sqlalchemy import create_engine

import matplotlib.pyplot as plt

import seaborn as sns

from datetime import datetime

import os



# Create directories for storing data

os.makedirs('data/raw', exist_ok=True)
os.makedirs('data/processed', exist_ok=True)


# Configuration

API_KEY = '93168025156f418290e67d20b533b60b'

BASE_URL = 'https://api.rawg.io/api/games'


# EXTRACT PHASE

def extract_game_data(num_pages=2):

    """

    Extract video game data from RAWG API.

    """

    print(f"Starting data extraction from RAWG API - {num_pages}
pages")


    games_list = []


    for page in range(1, num_pages + 1):

        print(f"Fetching page {page}...")


        params = {

            'key': API_KEY,

            'page': page,

            'page_size': 40,  # Max allowed per page
            'ordering': '-rating'  # Sort by rating descending
        }

        try:
            response = requests.get(BASE_URL, params=params)
            response.raise_for_status()  # Raise exception for bad
status codes

            data = response.json()

            for game in data['results']:
                # Safely extract ESRB rating
                esrb_rating = 'Not Rated'
                if game.get('esrb_rating'):
                    esrb_rating = game['esrb_rating'].get('name', 'Not
Rated')

                # Extract relevant game details
                game_data = {
                    'id': game.get('id'),
                    'name': game.get('name'),
                    'released': game.get('released'),
                    'rating': game.get('rating'),
                    'rating_top': game.get('rating_top', 5),

                    'ratings_count': game.get('ratings_count'),

                    'metacritic': game.get('metacritic'),

                    'playtime': game.get('playtime'),

                    'platforms': ',
'.join([platform['platform']['name'] for platform in
game.get('platforms', [])]),

                    'genres': ', '.join([genre['name'] for genre in
game.get('genres', [])]),

                    'stores': ', '.join([store['store']['name'] for
store in game.get('stores', [])]),

                    'tags': ', '.join([tag['name'] for tag in
game.get('tags', [])]),

                    'esrb_rating': esrb_rating,

                    'timestamp': datetime.now().strftime('%Y-%m-%d
%H:%M:%S')

                }

                games_list.append(game_data)



            # Respect API rate limits

            time.sleep(1)



        except requests.exceptions.RequestException as e:

            print(f"Error fetching page {page}: {e}")

            continue
```

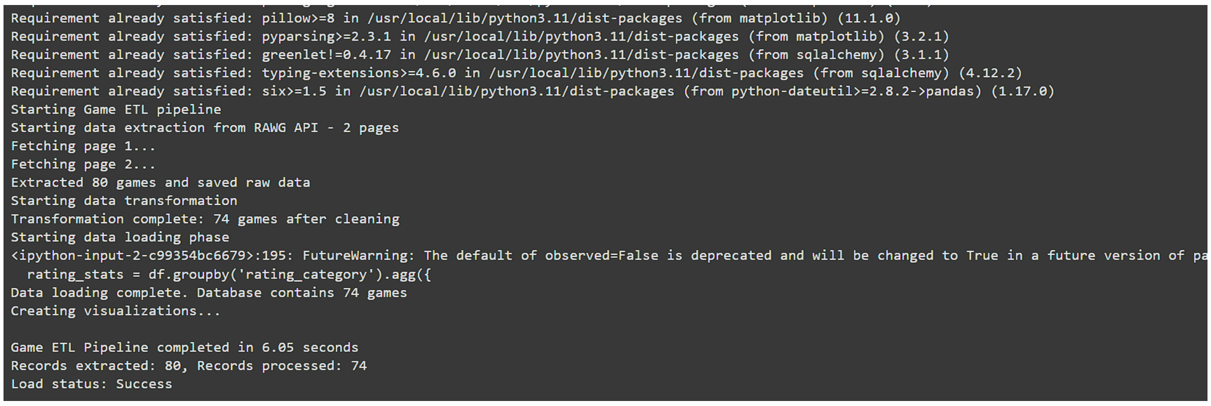
```python
    # Create DataFrame
    raw_df = pd.DataFrame(games_list)

    # Save raw data
    raw_df.to_csv('data/raw/rawg_raw_data.csv', index=False)
    print(f"Extracted {len(raw_df)} games and saved raw data")

    return raw_df

    # Create DataFrame
    raw_df = pd.DataFrame(games_list)

    # Save raw data
    raw_df.to_csv('data/raw/rawg_raw_data.csv', index=False)
    print(f"Extracted {len(raw_df)} games and saved raw data")

    return raw_df

# TRANSFORM PHASE
def transform_game_data(raw_df):
    """
    Clean and transform the raw game data.

    """

    print("Starting data transformation")


    # Create a copy to avoid modifying the original

    df = raw_df.copy()


    # 1. Convert date string to datetime and extract year

    df['released'] = pd.to_datetime(df['released'])

    df['release_year'] = df['released'].dt.year


    # 2. Calculate decade

    df['decade'] = (df['release_year'] // 10 * 10).astype('Int64')


    # 3. Normalize rating (since rating_top varies)

    df['normalized_rating'] = df['rating'] / df['rating_top']


    # 4. Create rating categories

    df['rating_category'] = pd.cut(df['normalized_rating'],

                                  bins=[0, 0.5, 0.7, 0.8, 0.9, 1.0],

                                  labels=['Poor', 'Average', 'Good',
'Great', 'Excellent'])
```

```
    # 5. Create playtime categories

    df['playtime_category'] = pd.cut(df['playtime'].fillna(0),

                                     bins=[-1, 0, 10, 20, 50,
float('inf')],

                                     labels=['Unknown', 'Short (<10h)',
'Medium (10-20h)', 'Long (20-50h)', 'Very Long (>50h)'])



    # 6. Create metacritic categories

    df['metacritic_category'] = pd.cut(df['metacritic'].fillna(0),

                                       bins=[0, 50, 70, 85, 100],

                                       labels=['Bad (<50)', 'Average
(50-70)', 'Good (70-85)', 'Great (85-100)'])



    # 7. Count number of platforms, genres, stores, tags

    df['platforms_count'] = df['platforms'].str.split(',').str.len()

    df['genres_count'] = df['genres'].str.split(',').str.len()

    df['stores_count'] = df['stores'].str.split(',').str.len()

    df['tags_count'] = df['tags'].str.split(',').str.len()



    # 8. Create popularity score (weighted combination of rating and
ratings count)

    df['normalized_ratings_count'] = (df['ratings_count'] -
df['ratings_count'].min()) / \

                                     (df['ratings_count'].max() -
df['ratings_count'].min())
```

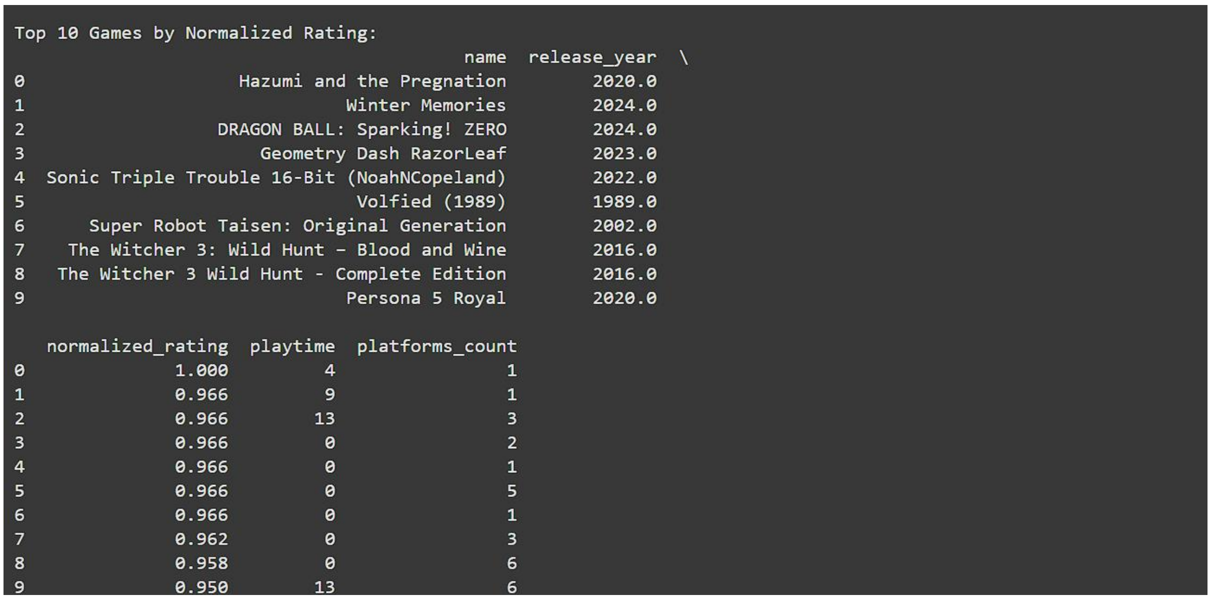
```
    df['popularity_score'] = (df['normalized_rating'] * 0.7) +
(df['normalized_ratings_count'] * 0.3)


    # 9. Clean ESRB ratings

    df['esrb_rating'] = df['esrb_rating'].replace('Not Rated',
'Unrated')


    # 10. Drop rows with missing essential data

    df = df.dropna(subset=['name', 'release_year', 'rating'])


    # 11. Select and reorder columns

    transformed_df = df[[

        'id', 'name', 'release_year', 'decade', 'released',

        'rating', 'normalized_rating', 'rating_category', 'rating_top',

        'ratings_count', 'metacritic', 'metacritic_category',

        'playtime', 'playtime_category',

        'platforms', 'platforms_count',

        'genres', 'genres_count',

        'stores', 'stores_count',

        'tags', 'tags_count',

        'esrb_rating', 'popularity_score', 'timestamp'

    ]]
    # Save transformed data

    transformed_df.to_csv('data/processed/rawg_transformed_data.csv',
index=False)

    print(f"Transformation complete: {len(transformed_df)} games after
cleaning")



    return transformed_df



# LOAD PHASE

def load_game_data(df):

    """

    Load the transformed data into a SQLite database.

    """

    print("Starting data loading phase")



    # Create SQLite database engine

    engine =
create_engine('sqlite:///data/processed/games_database.sqlite')



    # Create tables in the database

    # 1. Games table - main data

    df.to_sql('games', engine, if_exists='replace', index=False)



    # 2. Create some aggregated views as tables
    # Games by decade

    decade_stats = df.groupby('decade').agg({

        'name': 'count',

        'rating': 'mean',

        'playtime': 'mean',

        'popularity_score': 'mean',

        'metacritic': 'mean'

    }).reset_index()

    decade_stats.columns = ['decade', 'game_count', 'avg_rating',
'avg_playtime', 'avg_popularity', 'avg_metacritic']

    decade_stats.to_sql('decade_stats', engine, if_exists='replace',
index=False)



    # Games by rating category

    rating_stats = df.groupby('rating_category').agg({

        'name': 'count',

        'release_year': 'mean',

        'playtime': 'mean',

        'ratings_count': 'mean'

    }).reset_index()

    rating_stats.columns = ['rating_category', 'game_count',
'avg_release_year', 'avg_playtime', 'avg_ratings']

    rating_stats.to_sql('rating_stats', engine, if_exists='replace',
index=False)
```

```
    # Export to different formats
    decade_stats.to_csv('data/processed/decade_stats.csv', index=False)
    rating_stats.to_csv('data/processed/rating_stats.csv', index=False)

    # Verify data was loaded correctly
    conn = sqlite3.connect('data/processed/games_database.sqlite')
    game_count = pd.read_sql("SELECT COUNT(*) FROM games",
conn).iloc[0, 0]
    conn.close()


    print(f"Data loading complete. Database contains {game_count}
games")
    return True


# Visualize the data
def visualize_game_data(df):
    """
    Create visualizations of the game data.
    """
    print("Creating visualizations...")


    # Connect to database for some queries
    conn = sqlite3.connect('data/processed/games_database.sqlite')



    # Plot visualizations

    plt.figure(figsize=(18, 12))



    # Plot 1: Ratings distribution

    plt.subplot(2, 3, 1)

    sns.histplot(df['normalized_rating'], kde=True)

    plt.title('Distribution of Normalized Ratings')



    # Plot 2: Games by decade

    plt.subplot(2, 3, 2)

    decade_counts = df['decade'].value_counts().sort_index()

    sns.barplot(x=decade_counts.index.astype(str),
y=decade_counts.values)

    plt.title('Games by Decade')

    plt.xticks(rotation=45)



    # Plot 3: Rating vs. Playtime

    plt.subplot(2, 3, 3)

    sns.scatterplot(x='normalized_rating', y='playtime', data=df,
alpha=0.5)

    plt.title('Rating vs. Playtime')
    plt.xlabel('Normalized Rating')
    plt.ylabel('Playtime (hours)')


    # Plot 4: Average playtime by decade
    plt.subplot(2, 3, 4)
    playtime_by_decade = pd.read_sql("SELECT decade, avg_playtime FROM decade_stats ORDER BY decade", conn)
    sns.lineplot(x='decade', y='avg_playtime', data=playtime_by_decade, marker='o')
    plt.title('Average Playtime by Decade')
    plt.xlabel('Decade')
    plt.ylabel('Average Playtime (hours)')


    # Plot 5: ESRB Rating Distribution
    plt.subplot(2, 3, 5)
    esrb_counts = df['esrb_rating'].value_counts()
    sns.barplot(x=esrb_counts.index, y=esrb_counts.values)
    plt.title('ESRB Rating Distribution')
    plt.xticks(rotation=45)


    # Plot 6: Platforms Count Distribution
    plt.subplot(2, 3, 6)
    sns.histplot(df['platforms_count'], bins=range(1, df['platforms_count'].max()+2), discrete=True)
    plt.title('Distribution of Platforms per Game')

    plt.xlabel('Number of Platforms')



    plt.tight_layout()

    plt.savefig('data/processed/game_analysis_visualization.png')



    conn.close()



# Run the full ETL pipeline

def run_game_etl_pipeline(num_pages=2):

    """

    Execute the full ETL pipeline for game data.

    """

    start_time = time.time()

    print("Starting Game ETL pipeline")



    # Extract

    raw_data = extract_game_data(num_pages)



    # Transform

    transformed_data = transform_game_data(raw_data)
    # Simple data quality check

    if len(transformed_data) > 0:

        # Load

        load_success = load_game_data(transformed_data)



        # Create visualizations

        visualize_game_data(transformed_data)

    else:

        print("No data to load after transformation")

        load_success = False



    # Pipeline summary

    total_time = time.time() - start_time



    print(f"\nGame ETL Pipeline completed in {total_time:.2f} seconds")

    print(f"Records extracted: {len(raw_data)}, Records processed:
{len(transformed_data)}")

    print(f"Load status: {'Success' if load_success else 'Failed'}")



    return transformed_data



# Execute the ETL pipeline

# You can adjust the number of pages to fetch (each page has 40 games)
```

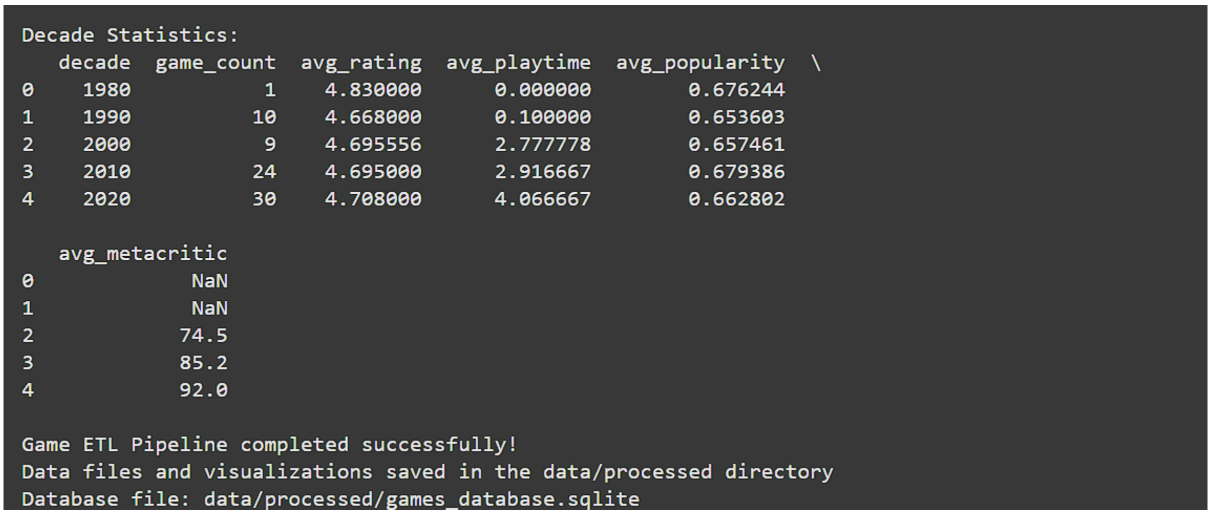
```
game_data = run_game_etl_pipeline(num_pages=2)



# Query and display some results from the database

conn = sqlite3.connect('data/processed/games_database.sqlite')



print("\nTop 10 Games by Normalized Rating:")

top_games = pd.read_sql("""

    SELECT name, release_year, normalized_rating, playtime,
platforms_count

    FROM games

    ORDER BY normalized_rating DESC

    LIMIT 10

""", conn)

print(top_games)



print("\nDecade Statistics:")

decade_stats = pd.read_sql("SELECT * FROM decade_stats ORDER BY
decade", conn)

print(decade_stats)



conn.close()



print("\nGame ETL Pipeline completed successfully!")
print("Data files and visualizations saved in the data/processed
directory")

print("Database file: data/processed/games_database.sqlite")
```



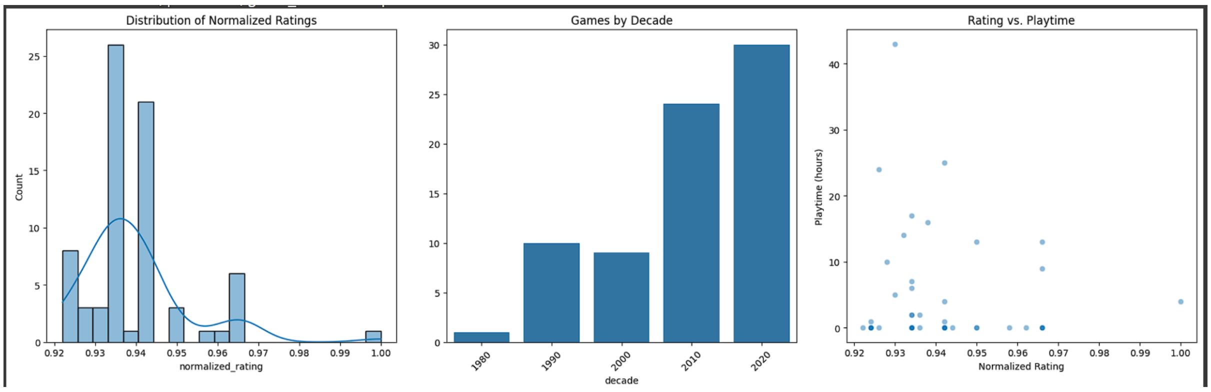
```
Requirement already satisfied: pillow>=8 in /usr/local/lib/python3.11/dist-packages (from matplotlib) (11.1.0)
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.11/dist-packages (from matplotlib) (3.2.1)
Requirement already satisfied: greenlet!=0.4.17 in /usr/local/lib/python3.11/dist-packages (from sqlalchemy) (3.1.1)
Requirement already satisfied: typing-extensions>=4.6.0 in /usr/local/lib/python3.11/dist-packages (from sqlalchemy) (4.12.2)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.11/dist-packages (from python-dateutil>=2.8.2->pandas) (1.17.0)
Starting Game ETL pipeline
Starting data extraction from RAWG API - 2 pages
Fetching page 1...
Fetching page 2...
Extracted 80 games and saved raw data
Starting data transformation
Transformation complete: 74 games after cleaning
Starting data loading phase
<ipython-input-2-c99354bc6679>:195: FutureWarning: The default of observed=False is deprecated and will be changed to True in a future version of pa
  rating_stats = df.groupby('rating_category').agg({
Data loading complete. Database contains 74 games
Creating visualizations...

Game ETL Pipeline completed in 6.05 seconds
Records extracted: 80, Records processed: 74
Load status: Success
```

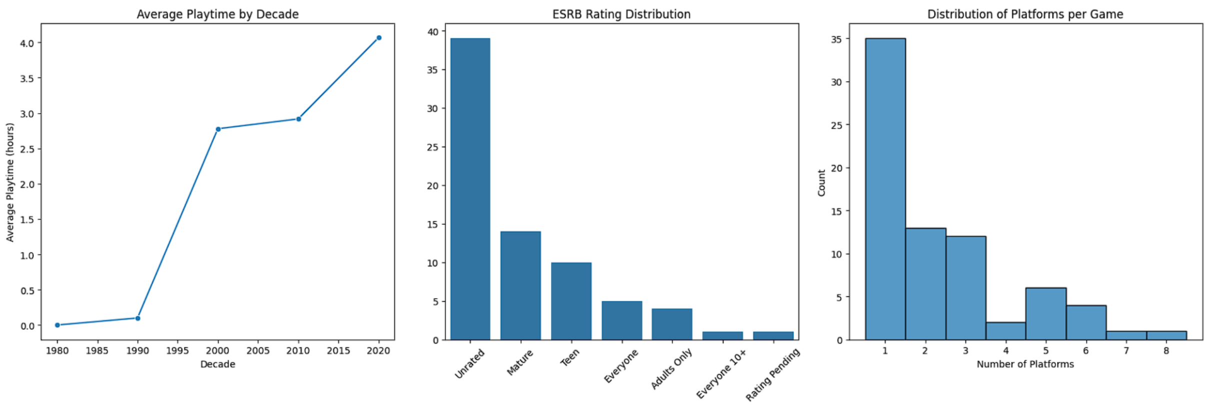


```
Top 10 Games by Normalized Rating:
                                          name  release_year  \
0                   Hazumi and the Pregnation        2020.0
1                             Winter Memories        2024.0
2                    DRAGON BALL: Sparking! ZERO        2024.0
3                        Geometry Dash RazorLeaf        2023.0
4  Sonic Triple Trouble 16-Bit (NoahNCopeland)        2022.0
5                                Volfied (1989)        1989.0
6          Super Robot Taisen: Original Generation        2002.0
7        The Witcher 3: Wild Hunt - Blood and Wine        2016.0
8      The Witcher 3 Wild Hunt - Complete Edition        2016.0
9                               Persona 5 Royal        2020.0

   normalized_rating  playtime  platforms_count
0              1.000         4                1
1              0.966         9                1
2              0.966        13                3
3              0.966         0                2
4              0.966         0                1
5              0.966         0                5
6              0.966         0                1
7              0.962         0                3
8              0.958         0                6
9              0.950        13                6
```



```
Decade Statistics:
   decade  game_count  avg_rating  avg_playtime  avg_popularity  \
0    1980           1    4.830000      0.000000        0.676244
1    1990          10    4.668000      0.100000        0.653603
2    2000           9    4.695556      2.777778        0.657461
3    2010          24    4.695000      2.916667        0.679386
4    2020          30    4.708000      4.066667        0.662802

   avg_metacritic
0             NaN
1             NaN
2            74.5
3            85.2
4            92.0

Game ETL Pipeline completed successfully!
Data files and visualizations saved in the data/processed directory
Database file: data/processed/games_database.sqlite
```





<u>Conclusion</u>:

This ETL pipeline successfully extracts video game data from the RAWG API, transforms it by cleaning, categorizing, and normalizing key features, and loads it into a structured SQLite database. Visualizations help analyze trends in ratings, playtime, and ESRB ratings across different decades.

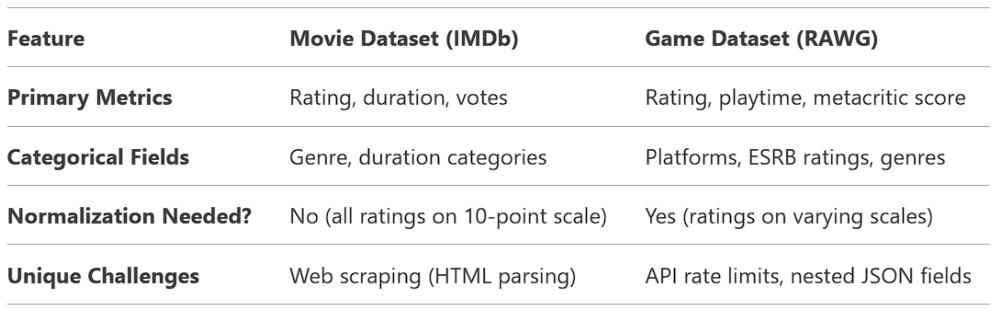
Post-lab questions:

1. What were the main challenges you faced while extracting and cleaning data from your chosen source? How did you handle missing or inconsistent data?
2. **API Limitations & Rate Throttling**: The RAWG API has rate limits, requiring delays (time.sleep) between requests to avoid bans.
3. **Inconsistent Data Formats**:
    a. ESRB ratings were sometimes missing or nested unpredictably (handled via .get() with fallbacks).
    b. playtime and metacritic scores had missing values (filled with defaults or categorized).
    c. Ratings used varying scales (e.g., some games had rating_top=5, others 10), requiring normalization.
4. **Text-Based Lists**: Fields like platforms and genres were comma-separated strings, needing splitting for analysis.

Handling Missing/Inconsistent Data:

5. **Default Values**: Used 'Not Rated' for missing ESRB ratings and 0 for missing playtime.
6. **Normalization**: Scaled ratings to a 0–1 range (normalized_rating = rating / rating_top) for fair comparison.
7. **Categorization**: Binned continuous variables (e.g., playtime_category, metacritic_category) to simplify analysis.
8. **Dropping Irrecoverable Data**: Removed rows missing critical fields like name or release_year.

2. How does the data from your chosen source compare with the movie dataset structure-wise?



Feature	Movie Dataset (IMDb)	Game Dataset (RAWG)
Primary Metrics	Rating, duration, votes	Rating, playtime, metacritic score
Categorical Fields	Genre, duration categories	Platforms, ESRB ratings, genres
Normalization Needed?	No (all ratings on 10-point scale)	Yes (ratings on varying scales)
Unique Challenges	Web scraping (HTML parsing)	API rate limits, nested JSON fields

3. If you had to process a much larger dataset (millions of records), what optimizations would you implement in your ETL pipeline? How would you ensure efficient storage and retrieval of data?

Extract Phase:

- **Parallel Requests**: Use async libraries like aiohttp or multiprocessing to fetch data concurrently.
- **Incremental Loading**: Track timestamps to only fetch new/updated records (avoid full extracts).

Transform Phase:

- **Chunking**: Process data in batches (e.g., Pandas chunksize) to reduce memory usage.
- **Vectorized Operations**: Replace loops with Pandas/Numpy operations (e.g., .apply() → .map()).
- **Dask/Spark**: For distributed processing of very large datasets.

Load Phase:

- **Database Optimizations**:
    - Use **PostgreSQL** or **BigQuery** (instead of SQLite) for scalability.
    - Add **indexes** on frequently queried columns (e.g., release_year).
    - Partition tables by decade/genre for faster queries.
- **Columnar Storage**: Save processed data as **Parquet** (optimized for analytics) instead of CSV.

Efficient Storage/Retrieval:

- **Data Warehousing**: Load into **Snowflake** or **Redshift** for OLAP workloads.
- **Caching**: Cache aggregated results (e.g., decade stats) to avoid recomputation.