**Data Science**

**Final Project**

**Game Lens**

**An Analysis of Steam Games’ Data**

**Project Report**

**Dated: 17 December 2024**

**Projects Members:**

**Mohammed Ali Khowaja (2112150)**

**Muhammad Hamza Javed (2112153)**

**Om Kirshana (2112283)**

**Manoj Kumar (2112147)**

**Asfund Ali Amjad (2112341)**

**Abstract:**

This report examines the game data from SteamDB, SteamCharts, SteamSpy, and the Steam API to identify trends and insights in game performance, player demographics, and market dynamics. The data collection was strategically divided among four team members to fetch a total of 20,000 game entries. The primary objective of this project is to analyze gaming trends, user preferences, and market dynamics by leveraging the dataset. The scope includes identifying key patterns, providing actionable insights, and creating an interactive dashboard for data exploration. This project can benefit game developers, publishers, and analysts by aiding in decision-making processes.

**Introduction:**

Steam is the largest digital distribution platforms for PC gaming, hosting thousands of titles across various genres and platforms. Started in September of 2003, it began as platform by Valve to distribute its video games, and later began to sell the rights to play the games from independent developers and major distributors and has become the largest distributor.

By 2011, Steam has approximately 50-70% of the market for downloadable PC games, with a userbase of about 40 million accounts. Analyzing game data from Steam provides valuable insights into market trends, player engagement, and business strategies. This report explores key metrics such as game popularity, pricing strategies, and platform support to uncover patterns and opportunities.

The purpose of this project is to perform data science workflows on a dataset containing information about various games. To ensure a manageable and comprehensive dataset, we limited our collection to 20,000 games, distributing the task among four members to split the workload evenly. This report provides an overview of the data attributes, data collection and cleaning methods, and a preliminary plan for analysis.

**Dataset Description:**

The dataset consists of the following columns, each representing an important attribute of the games:

1. appid: Unique game ID.
2. release\_date: Launch date.
3. required\_age: Minimum player age.
4. dlc\_count: Number of DLCs (Downloadable Content) available.
5. windows, mac, linux: Platform availability.
6. categories: Gameplay features (e.g., multiplayer, co-op).
7. genres: Game genres in a JSON format array (e.g., Action, RPG, Racing).
8. name, developer, publisher: Game details.
9. owners: Estimated ownership range.
10. price, initialprice: Pricing details in cents.
11. Discounts: Discounts rates in percentage.
12. all\_time\_peak\_ccu: Peak concurrent user count.
13. technologies: Game engines or frameworks.

**Methodology:**

**Data Scraping:**

We first identified a set of websites where we can extract the data from and then made the preliminary list of fields to extract from each site.

* **SteamDB:** For ownership, user activity metrics, platforms and technologies utilised in the video game.
* **SteamCharts:** For the all-time peak concurrent player count.
* **SteamSpy:** For ownership, price, initial price and discount fields.
* **Steam API:** For required age, DLC (downloadable content) count.

Our data scraping consisted of using tools such as:

* **Cheerio**: for HTML parsing.
* **Axios**: for scraping and HTTP requests.
* **Puppeteer**: for scraping in a Chrome -like environment to extract data from the sources.

All our data was fetched and exported into a JSON file format, merged and then exported into a CSV for Power BI and Python analysis.

**Ethical Considerations:**

This project adheres to ethical web scraping practices by respecting user privacy and ensuring no personal data is collected and complying with terms of service of the data sources.

**Validation Processes:**

Before we began scraping data, to ensure that the data reliability, and to ensure that there are no errors in the extracted data, we conducted a test run that went through the following validation process:

* **Consistency Checks**: Verifying data consistency over time to detect anomalies.
* **Cross-Validation**: Comparing data against other sources to ensure data is correct.
* **Unit Testing**: Testing scraping and cleaning scripts to identify and resolve bugs.

### Data Filtering and Cleaning:

After determining the set of characteristics to extract, our data was mostly clean and required very little cleaning.

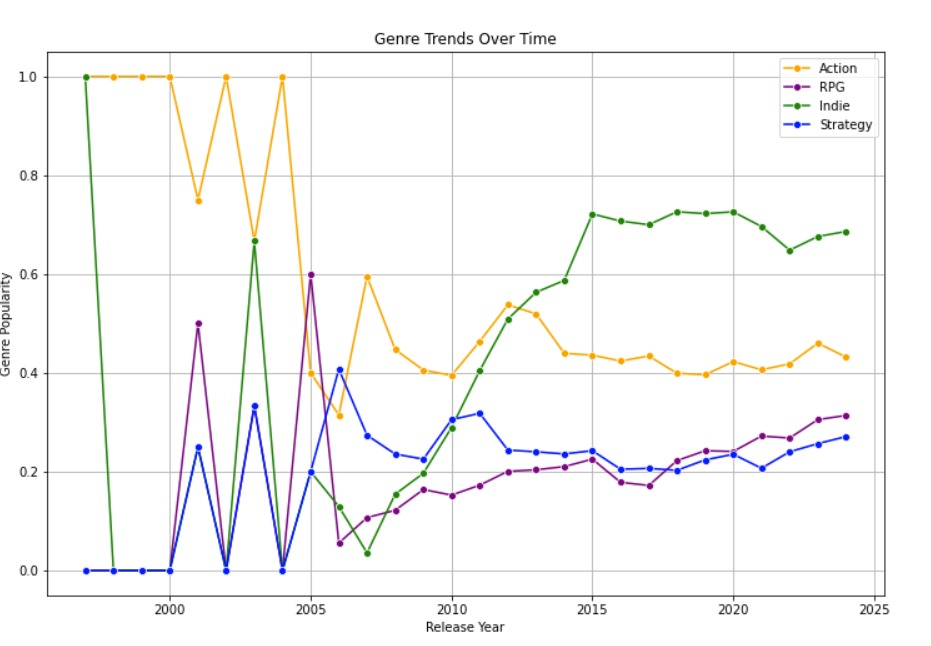
**Steps Taken:**

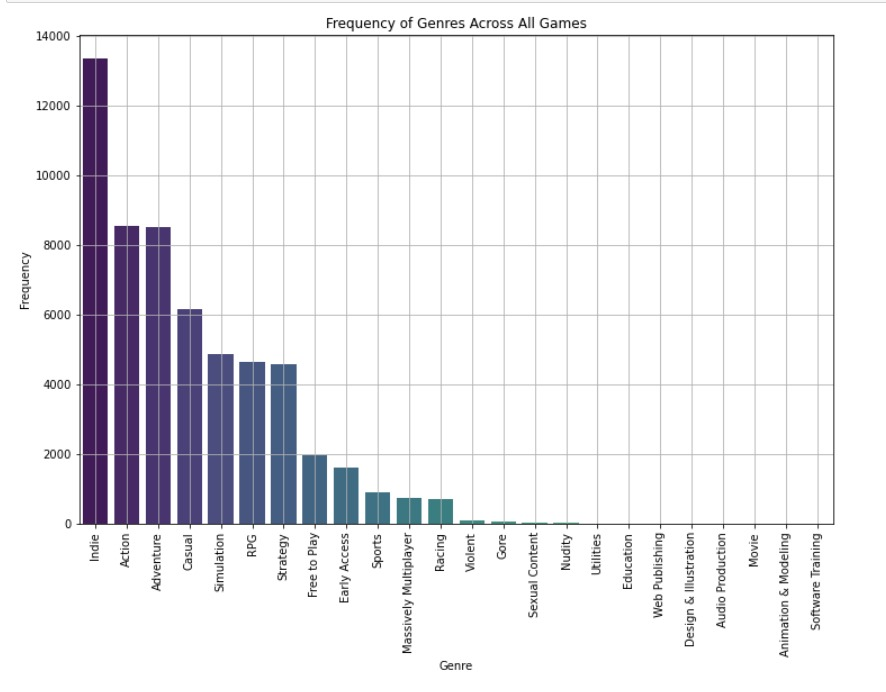
* Converted all release dates into the same DD/MM/YYYY format.
* Stripping whitespace and removing special characters in text fields.
* Converted pricing in cents to dollars, using Power BI DAX Query.
* Extracted different genres into their own Boolean columns.

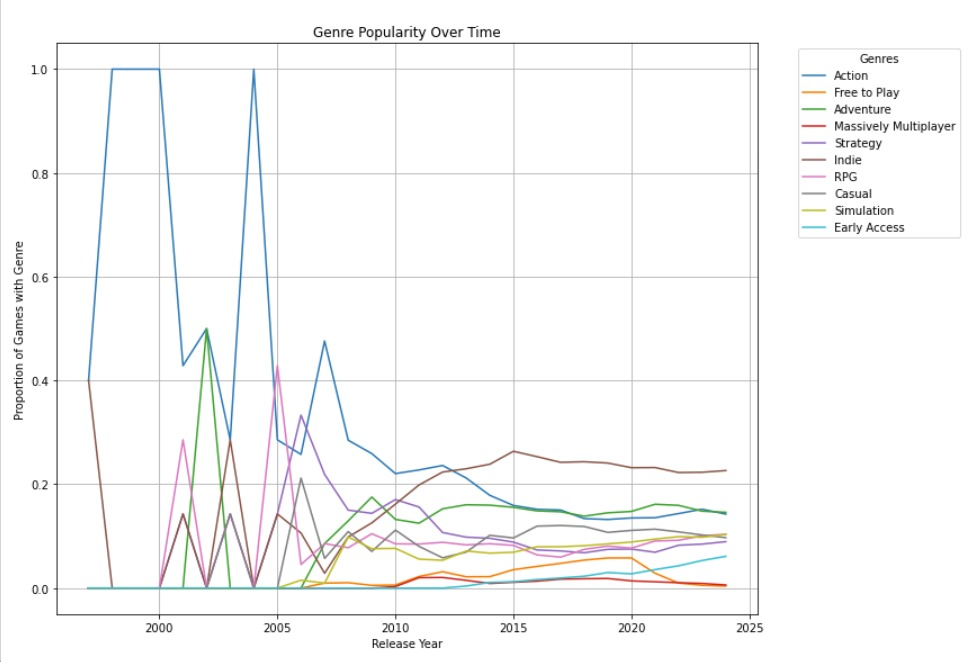
### Visualizations and Analysis:

**Python Models:**

Using Python libraries such as Pandas, Matploblib and others, we performed some data visualizations and analysis. The goal was to analyze the occurrence of different genres and their popularity over time.







**Code:**

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

# Load dataset (replace with your actual dataset file path)

data = pd.read\_csv('cleaned\_data.csv')

# Convert 'release\_date' to datetime format

data['release\_date'] = pd.to\_datetime(data['release\_date'], errors='coerce')

# Clean dataset by filling missing values (if needed)

data.fillna(0, inplace=True)

# Extract year from the 'release\_date' for trend analysis

data['release\_year'] = data['release\_date'].dt.year

# Filter out rows where the 'release\_year' is not valid

data = data[data['release\_year'] > 0]

# List of genre columns

genre\_columns = [

'Action', 'Free to Play', 'Adventure', 'Massively Multiplayer', 'Strategy',

'Indie', 'RPG', 'Casual', 'Simulation', 'Early Access', 'Sports', 'Racing',

'Violent', 'Gore', 'Movie', 'Animation & Modeling', 'Design & Illustration',

'Education', 'Software Training', 'Utilities', 'Web Publishing', 'Sexual Content',

'Nudity', 'Audio Production'

]

# Frequency analysis of genres: Count the number of games with each genre

genre\_frequency = data[genre\_columns].sum()

# Plot genre frequency distribution

plt.figure(figsize=(12, 8))

genre\_frequency\_sorted = genre\_frequency.sort\_values(ascending=False)

sns.barplot(x=genre\_frequency\_sorted.index, y=genre\_frequency\_sorted.values, palette='viridis')

plt.title('Frequency of Genres Across All Games')

plt.xlabel('Genre')

plt.ylabel('Frequency')

plt.xticks(rotation=90)

plt.grid(True)

plt.show()

# Track genre popularity over time (e.g., by release year)

# Aggregate the occurrence of each genre by year

genre\_trends = data.groupby('release\_year')[genre\_columns].sum()

# Normalize to get the proportion of each genre per year

genre\_trends\_normalized = genre\_trends.div(genre\_trends.sum(axis=1), axis=0)

# Plot genre trends over time (first few popular genres)

plt.figure(figsize=(12, 8))

for genre in genre\_columns[:10]: # Plot the first 10 genres for better visualization

sns.lineplot(data=genre\_trends\_normalized, x=genre\_trends\_normalized.index, y=genre, label=genre)

plt.title('Genre Popularity Over Time')

plt.xlabel('Release Year')

plt.ylabel('Proportion of Games with Genre')

plt.legend(title='Genres', bbox\_to\_anchor=(1.05, 1), loc='upper left')

plt.grid(True)

plt.tight\_layout()

plt.show()

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

# Load dataset (replace with your actual dataset file path)

data = pd.read\_csv('cleaned\_data.csv')

# Convert 'release\_date' to datetime format

data['release\_date'] = pd.to\_datetime(data['release\_date'], errors='coerce')

# Clean dataset by filling missing values (if needed)

data.fillna(0, inplace=True)

# Extract year from the 'release\_date' for trend analysis

data['release\_year'] = data['release\_date'].dt.year

# Filter out rows where the 'release\_year' is not valid

data = data[data['release\_year'] > 0]

# List of genre columns

genre\_columns = [

'Action', 'Free to Play', 'Adventure', 'Massively Multiplayer', 'Strategy',

'Indie', 'RPG', 'Casual', 'Simulation', 'Early Access', 'Sports', 'Racing',

'Violent', 'Gore', 'Movie', 'Animation & Modeling', 'Design & Illustration',

'Education', 'Software Training', 'Utilities', 'Web Publishing', 'Audio Production'

]

# Frequency analysis of genres: Count the number of games with each genre

genre\_frequency = data[genre\_columns].sum()

# Plot genre frequency distribution

plt.figure(figsize=(12, 8))

genre\_frequency\_sorted = genre\_frequency.sort\_values(ascending=False)

sns.barplot(x=genre\_frequency\_sorted.index, y=genre\_frequency\_sorted.values, palette='viridis')

plt.title('Frequency of Genres Across All Games')

plt.xlabel('Genre')

plt.ylabel('Frequency')

plt.xticks(rotation=90)

plt.grid(True)

plt.show()

# Track genre popularity over time (e.g., by release year)

# Aggregate the occurrence of each genre by year

genre\_trends = data.groupby('release\_year')[genre\_columns].sum()

# Normalize to get the proportion of each genre per year

genre\_trends\_normalized = genre\_trends.div(genre\_trends.sum(axis=1), axis=0)

# Plot genre trends over time (first few popular genres)

plt.figure(figsize=(12, 8))

for genre in genre\_columns[:10]: # Plot the first 10 genres for better visualization

sns.lineplot(data=genre\_trends\_normalized, x=genre\_trends\_normalized.index, y=genre, label=genre)

plt.title('Genre Popularity Over Time')

plt.xlabel('Release Year')

plt.ylabel('Proportion of Games with Genre')

plt.legend(title='Genres', bbox\_to\_anchor=(1.05, 1), loc='upper left')

plt.grid(True)

plt.tight\_layout()

plt.show()

**Power BI:**

Using Power BI, the following visualizations were created:

### General: The general page shows the All-Time Peak Concurrent Users of the games, the number of games, estimated ownership of the game, a pie chart showing the games availability by platform (Operating System) and a scatter plot comparing DLC counts to peak user engagement.

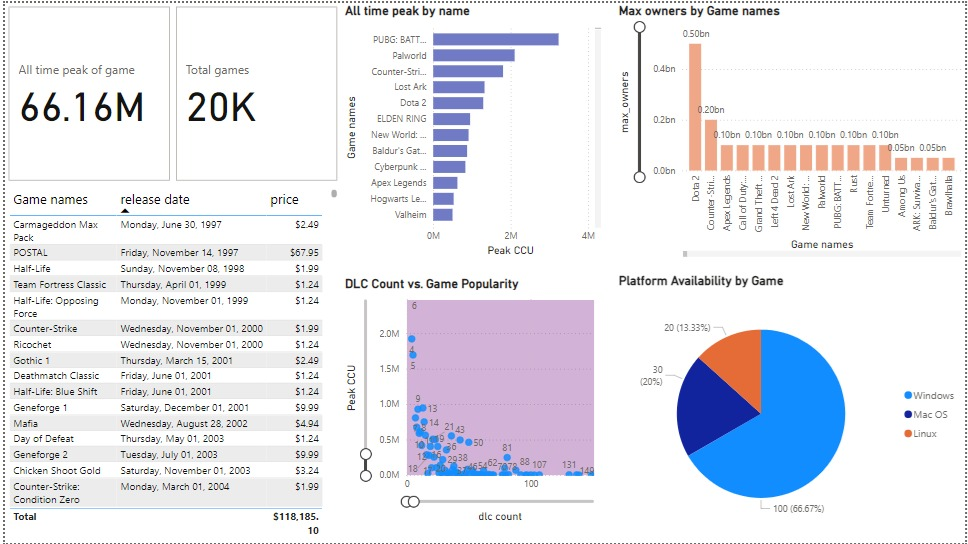
### Game Popularity by Genre: In this page, we investigated games popularity by genre and more. With the help of filters (slicers in Power BI), we have made it interactive where you can easily see the top games of the genre.

### Overall Game Popularity: In this page, we created a tree map visualization to highlight games with high player engagement since their release.

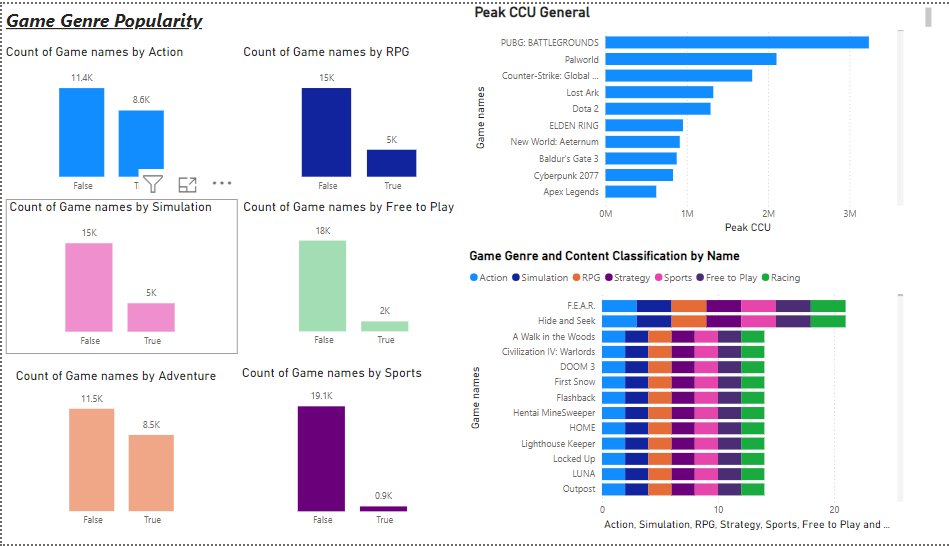
### Pricing: In this page, we investigated games pricing trends and more.

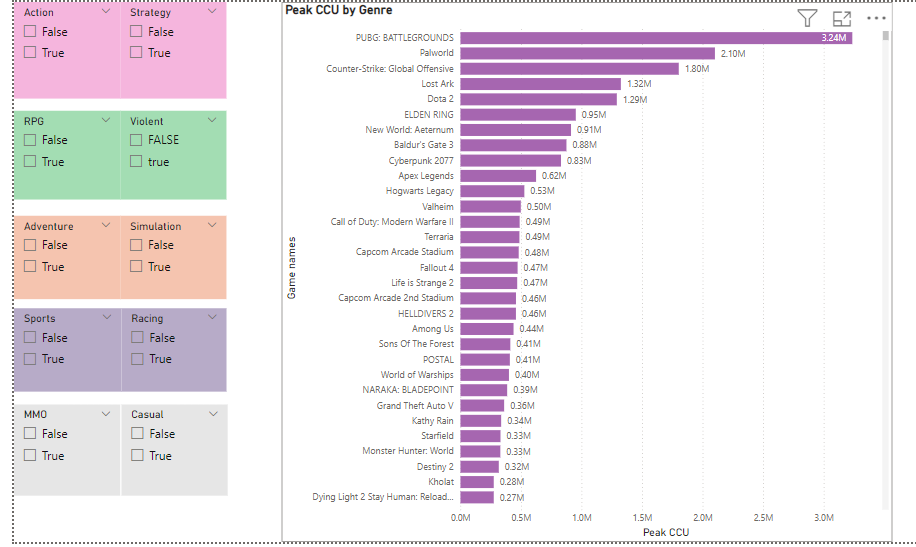
### Dashboard:

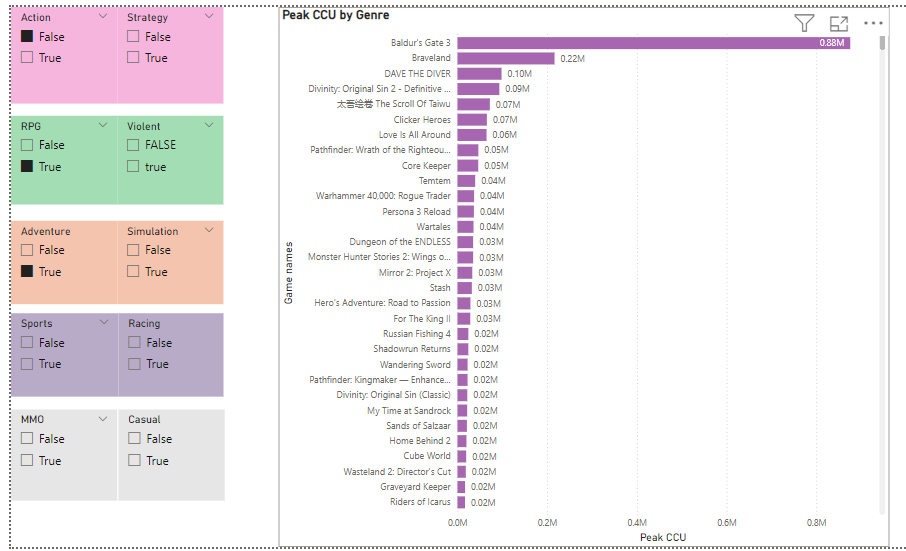
* General:



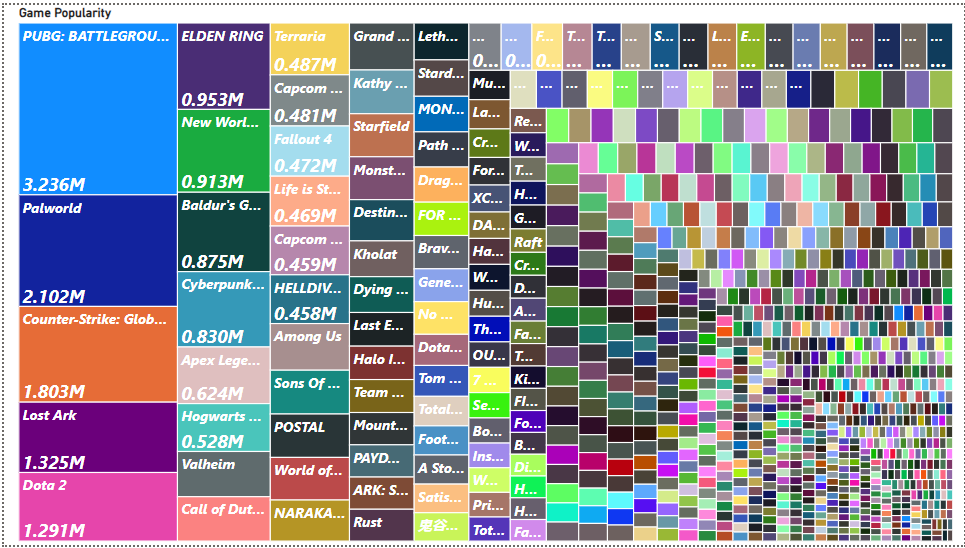
* Game Popularity by Genre:

****

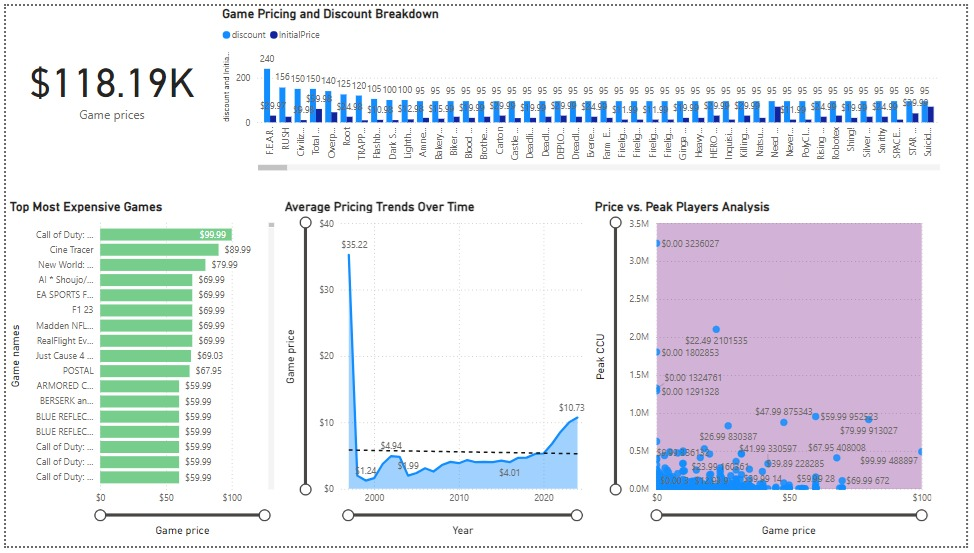
****

****

* Overall Game Popularity:

****

* Pricing:



**Limitations:**

Acknowledged limitations include:

* **Data Completeness**: Potential gaps in scraped data due to restricted access and a smaller size of the dataset.
* **Assumptions**: Reliance on estimated metrics like ownership counts.

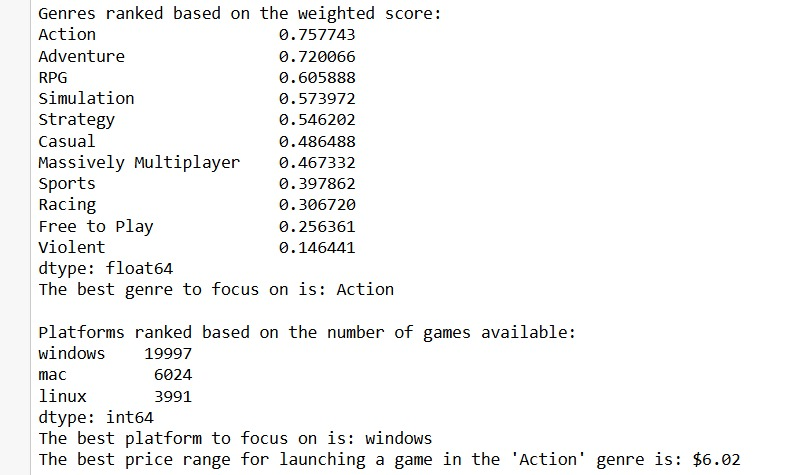
**Machine Learning Models:**

We developed two models on our dataset using Python libraries:

* Best Genre with Pricing Recommendation.
* Personalized Game Recommendation.
* Game Success Predictor.

**Best Genre with Pricing Recommendation Model:**

* **Task**: Suggest the best genre, platform, and price for launching a new game based on historical data analysis.
* **Outcome:**
  + Best Genre: Top-ranked genre based on a combination of factors.
  + Best Platform: Platform with the most games, indicating high demand.
  + Best Price: Suggested price for a new game, adjusted for industry trends.
* **Screenshot:**

****

* **Code:**

import pandas as pd

# Load the dataset

data = pd.read\_csv('result.csv')

# Remove the unwanted columns

columns\_to\_remove = ['Early Access', 'Indie', 'Gore', 'Movie', 'Animation & Modeling', 'Design & Illustration', 'Education', 'Software Training', 'Utilities', 'Web Publishing', 'Audio Production']

data.drop(columns=columns\_to\_remove, axis=1, inplace=True)

# Clean the data (remove rows with missing genres or ratings if applicable)

data.dropna(subset=['genres'], inplace=True)

# Count how many games belong to each genre (by summing the genre columns)

genre\_columns = ['Action', 'Free to Play', 'Adventure', 'Massively Multiplayer', 'Strategy', 'RPG', 'Casual', 'Simulation', 'Sports', 'Racing', 'Violent']

# Count the number of games in each genre

genre\_counts = data[genre\_columns].sum().sort\_values(ascending=False)

# Find the average price for games in each genre

genre\_avg\_price = data[genre\_columns].multiply(data['price'], axis=0).sum() / genre\_counts

# Find the average all-time peak concurrent users for each genre

genre\_avg\_ccu = data[genre\_columns].multiply(data['all\_time\_peak\_ccu'], axis=0).sum() / genre\_counts

# Now create a scoring model based on the above features

# Normalize the features (to make them comparable)

normalized\_game\_count = genre\_counts / genre\_counts.max()  # Normalize based on max game count

normalized\_avg\_price = genre\_avg\_price / genre\_avg\_price.max()  # Normalize price

normalized\_avg\_ccu = genre\_avg\_ccu / genre\_avg\_ccu.max()  # Normalize peak concurrent users

# Assign weights to each feature (you can adjust these weights based on your business goal)

weight\_game\_count = 0.4

weight\_avg\_price = 0.3

weight\_avg\_ccu = 0.3

# Calculate the total score for each genre

genre\_scores = (normalized\_game\_count \* weight\_game\_count) + \

               (normalized\_avg\_price \* weight\_avg\_price) + \

               (normalized\_avg\_ccu \* weight\_avg\_ccu)

# Convert genre\_scores to numeric explicitly in case there are any non-numeric entries

genre\_scores = pd.to\_numeric(genre\_scores, errors='coerce')

# Drop NaN values that might result from the coercion

genre\_scores = genre\_scores.dropna()

# Rank genres based on their total score

ranked\_genres = genre\_scores.sort\_values(ascending=False)

# Display the genres and their scores

print("Genres ranked based on the weighted score:")

print(ranked\_genres)

# Suggest the best genre to focus on, checking if the series is non-empty

if not ranked\_genres.empty:

    best\_genre = ranked\_genres.idxmax()  # Genre with the highest total score

    print(f"The best genre to focus on is: {best\_genre}")

else:

    print("No valid genres to suggest.")

# Suggest which platform to focus on (Mac, Windows, Linux)

platform\_columns = ['windows', 'mac', 'linux']

# Count the number of games available for each platform

platform\_counts = data[platform\_columns].sum().sort\_values(ascending=False)

# Display the platforms and their game counts

print("\nPlatforms ranked based on the number of games available:")

print(platform\_counts)

# Suggest the best platform to focus on, checking if the platform count series is non-empty

if not platform\_counts.empty:

    best\_platform = platform\_counts.idxmax()  # Platform with the highest game count

    print(f"The best platform to focus on is: {best\_platform}")

else:

    print("No valid platform to suggest.")

# Suggest the best price based on the genre ranking

# The average price for the top genre

best\_genre\_avg\_price = genre\_avg\_price[best\_genre]

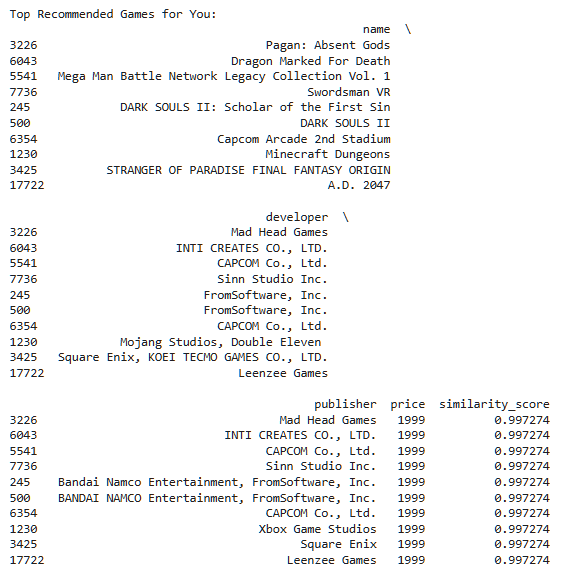
# Divide the best price by 100 to scale it down

best\_genre\_avg\_price /= 100

print(f"The best price range for launching a game in the '{best\_genre}' genre is: ${best\_genre\_avg\_price:.2f}")

**Personalized Game Recommendation Model:**

* **Task:** The task involves recommending similar games to a user based on a given game name. The algorithm uses Cosine Similarity to measure the similarity between the features of the provided game and all other games in the dataset. Features such as genre, price, developer, and publisher are used to create a vector representation of each game.
* **Outcome:** The outcome of the algorithm is a list of the top 10 most similar games to the input game. The recommendations are sorted by their similarity score, with higher scores indicating games that share more similar characteristics to the provided game. This allows users to discover games that are closely aligned with the one they input.
* **Screenshot:**

****

* **Code:**

import pandas as pd

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.preprocessing import MinMaxScaler

from sklearn.metrics.pairwise import cosine\_similarity

import numpy as np

import json

# Load Dataset

df = pd.read\_csv('games.csv')

# Drop unnecessary columns

columns\_to\_drop = ['Movie', 'Animation & Modeling', 'Design & Illustration', 'Software Training',

                   'Utilities', 'Web Publishing', 'Sexual Content', 'Nudity', 'Audio Production']

df = df.drop(columns=columns\_to\_drop, errors='ignore')

# Data Cleaning Function

def clean\_data(data):

    # List of boolean columns for conversion

    bool\_columns = [

        'Action', 'Free to Play', 'Adventure', 'Massively Multiplayer', 'Strategy',

        'Indie', 'RPG', 'Casual', 'Simulation', 'Early Access', 'Sports',

        'Racing', 'Violent', 'Gore', 'Education'

    ]

    # Convert boolean columns from text to 1 (True) or 0 (False)

    for col in bool\_columns:

        data[col] = data[col].apply(lambda x: 1 if str(x).lower() in ['true', '1', 'yes'] else 0)

    # Handle missing values for boolean columns (convert NaNs to 0)

    data[bool\_columns] = data[bool\_columns].fillna(0)

    # Handle missing textual data (developer/publisher)

    data['developer'] = data['developer'].fillna('Unknown')

    data['publisher'] = data['publisher'].fillna('Unknown')

    return data

df = clean\_data(df)

# Get User Preferences

def get\_user\_preferences():

    print("Enter your preferences for the following features:")

    user\_preferences = {}

    bool\_columns = [

        'Action', 'Free to Play', 'Adventure', 'Massively Multiplayer', 'Strategy',

        'Indie', 'RPG', 'Casual', 'Simulation', 'Early Access', 'Sports',

        'Racing', 'Violent', 'Gore', 'Education'

    ]

    for col in bool\_columns:

        while True:

            try:

                user\_input = int(input(f"Do you like {col}? (1 for Yes, 0 for No): "))

                if user\_input in [0, 1]:

                    user\_preferences[col] = user\_input

                    break

                else:

                    print("Please enter 1 for Yes or 0 for No.")

            except ValueError:

                print("Invalid input, please enter 1 for Yes or 0 for No.")

    # Price preference

    while True:

        try:

            max\_price = float(input("Enter the maximum price you are willing to pay (in dollars): "))

            user\_preferences['max\_price'] = max\_price

            break

        except ValueError:

            print("Invalid input, please enter a valid price in dollars.")

    # Toggle developer/publisher similarity

    while True:

        include\_developer\_publisher = input("Include developer/publisher in recommendations? (yes/no): ").strip().lower()

        if include\_developer\_publisher in ['yes', 'no']:

            user\_preferences['use\_dev\_pub'] = include\_developer\_publisher == 'yes'

            break

        else:

            print("Please answer 'yes' or 'no'.")

    return user\_preferences

# Feature Matrix Construction

def create\_feature\_matrix(data, user\_preferences):

    # List of boolean columns for features

    bool\_columns = [

        'Action', 'Free to Play', 'Adventure', 'Massively Multiplayer', 'Strategy',

        'Indie', 'RPG', 'Casual', 'Simulation', 'Early Access', 'Sports',

        'Racing', 'Violent', 'Gore', 'Education'

    ]

    # Build the user vector from their preferences (genres + max price)

    user\_vector = np.array([user\_preferences[col] for col in bool\_columns] + [user\_preferences['max\_price']])

    # Create the base feature matrix (genre + price)

    feature\_matrix = data[bool\_columns + ['price']].values

    user\_price\_vector = np.full(len(data), user\_preferences['max\_price'])

    # Handle optional developer/publisher embedding

    dev\_pub\_matrix = None

    if user\_preferences['use\_dev\_pub']:

        vectorizer = TfidfVectorizer()

        dev\_pub\_matrix = vectorizer.fit\_transform(data['developer'] + " " + data['publisher'])

    return feature\_matrix, dev\_pub\_matrix, user\_vector, user\_price\_vector

# Recommendation Engine

def recommend\_games(data, user\_preferences):

    # Filter by minimum owners (ensure the game has a sufficient number of owners)

    filtered\_data = data[data['min\_owners'] > 30000]

    # Convert user's max price from dollars to cents

    max\_price\_in\_cents = user\_preferences['max\_price'] \* 100

    # Apply price filter (only games with price <= max\_price\_in\_cents)

    filtered\_data = filtered\_data[filtered\_data['price'] <= max\_price\_in\_cents]

    # Create feature matrix and user vector

    feature\_matrix, dev\_pub\_matrix, user\_vector, user\_price\_vector = create\_feature\_matrix(filtered\_data, user\_preferences)

    # Calculate price similarity (difference in price)

    price\_diff = np.abs(filtered\_data['price'].values - max\_price\_in\_cents)  # Compare prices in cents

    price\_similarity = 1 - (price\_diff / max\_price\_in\_cents)  # Compare in cents, clamp to [0, 1]

    # Calculate genre similarity using cosine similarity

    genre\_similarity = cosine\_similarity(feature\_matrix, user\_vector.reshape(1, -1)).flatten()

    # Combine genre and price similarity scores (adjust weights as needed)

    similarity\_scores = (0.7 \* genre\_similarity + 0.3 \* price\_similarity)

    # Include developer/publisher similarity if enabled

    if dev\_pub\_matrix is not None:

        dev\_pub\_similarity = cosine\_similarity(dev\_pub\_matrix, dev\_pub\_matrix[0].reshape(1, -1)).flatten()

        similarity\_scores += 0.5 \* dev\_pub\_similarity

    # Add similarity score to filtered data

    filtered\_data['similarity\_score'] = similarity\_scores

    # Sort by similarity and get top recommendations

    recommendations = filtered\_data.sort\_values(by='similarity\_score', ascending=False).head(10)

    return recommendations

# Main Function to run the recommendation system

def main():

    # user\_preferences = get\_user\_preferences()

    user\_preferences = {

        'Action': 1,

        'Free to Play': 1,

        'Adventure': 0,

        'Massively Multiplayer': 0,

        'Strategy': 0,

        'Indie': 0,

        'RPG': 1,

        'Casual': 0,

        'Simulation': 0,

        'Early Access': 0,

        'Sports': 0,

        'Racing': 0,

        'Violent': 0,

        'Gore': 0,

        'Education': 0,

        'max\_price': 20.0,  # Maximum price willing to pay in dollars

        'use\_dev\_pub': False  # Whether to include developer/publisher in recommendations

    }

    recommendations = recommend\_games(df, user\_preferences)

    print("\nTop Recommended Games for You:")

    print(recommendations[['name', 'developer', 'publisher', 'price', 'similarity\_score']])

if \_\_name\_\_ == "\_\_main\_\_":

    main()

**Game Success Predictor:**

* **Task:** The model predicts the game’s success based on the category, genre, game OS, as well as based on the user preferences, and outputs peak CCU, ownership and success status.
* **Outcome:** The outcome of the algorithm is that it shows the accuracy of the prediction, as well as the status whether your game will be successful or not given on the category, genre and platform.
* **Screenshot:**





* **Code:**

import pandas as pd

from sklearn.ensemble import RandomForestClassifier

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import accuracy\_score

# Load dataset (replace with your actual dataset file path)

data = pd.read\_csv('cleaned\_data.csv')

# Clean dataset: Fill missing values with 0 (or other appropriate strategies)

data.fillna(0, inplace=True)

# Define the columns for input features (as before)

genre\_columns = [ 'Action', 'Free to Play', 'Adventure', 'Massively Multiplayer', 'Strategy',

'Indie', 'RPG', 'Casual', 'Simulation', 'Early Access', 'Sports', 'Racing',

'Violent', 'Gore', 'Education'

]

category\_columns = [ 'category\_Single-player', 'category\_Multi-player', 'category\_Co-op',

'category\_Online\_PvP', 'category\_VR\_Support', 'category\_Steam\_Cloud',

'category\_Steam\_Trading\_Cards', 'category\_Steam\_Workshop', 'category\_LAN\_Co-op',

'category\_Steam\_Achievements', 'category\_Cross-Platform\_Multiplayer',

'category\_In-App\_Purchases', 'category\_Remote\_Play\_on\_Tablet', 'category\_VR\_Only'

]

os\_columns = ['windows', 'mac', 'linux']

# Numerical features for prediction

numerical\_columns = ['price', 'dlc\_count']

# Combine all relevant features for prediction

all\_features = genre\_columns + category\_columns + os\_columns + numerical\_columns

# Define success based on a threshold for peak CCU or owners\_midpoint

success\_threshold = 1000000 # Example threshold

data['is\_successful'] = (data['all\_time\_peak\_ccu'] > success\_threshold) | (data['owners\_midpoint'] > success\_threshold)

# Features and target variable for classification

X = data[all\_features]

y = data['is\_successful']

# Split the dataset into training and testing sets for classification

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Train the Random Forest Classifier

clf = RandomForestClassifier(random\_state=42)

clf.fit(X\_train, y\_train)

# Evaluate the model (optional)

y\_pred = clf.predict(X\_test)

classification\_accuracy = accuracy\_score(y\_test, y\_pred)

print(f"Model Accuracy: {classification\_accuracy:.4f}")

# Function to predict the success of a new game based on user input

def predict\_game\_success(user\_input):

"""

Predict the success of a game based on user input.

user\_input should be a dictionary with keys as features and values as True/False.

"""

# Convert user input into a DataFrame

user\_input\_df = pd.DataFrame([user\_input])

# Ensure the DataFrame has the same columns as the training data

missing\_cols = set(X.columns) - set(user\_input\_df.columns)

for col in missing\_cols:

user\_input\_df[col] = 0 # Add missing columns with default value 0 (False)

# Make a prediction using the trained model

prediction = clf.predict(user\_input\_df)

return "Success" if prediction[0] == 1 else "Unsuccessful"

# Example of how the developer would input game attributes

print("Please input the following features for your new game:")

user\_input = {

'Action': True,

'Free to Play': False,

'Adventure': False,

'Massively Multiplayer': False,

'Strategy': False,

'Indie': False,

'RPG': False,

'Casual': False,

'Simulation': False,

'Early Access': False,

'Sports': False,

'Racing': False,

'Violent': True,

'Gore': True,

'Education': False,

'category\_Single-player': False,

'category\_Multi-player': True,

'category\_Co-op': True,

'category\_Online\_PvP': False,

'category\_VR\_Support': False,

'category\_Steam\_Cloud': True,

'category\_Steam\_Trading\_Cards': True,

'category\_Steam\_Workshop': False,

'category\_LAN\_Co-op': False,

'category\_Steam\_Achievements': True,

'category\_Cross-Platform\_Multiplayer': False,

'category\_In-App\_Purchases': False,

'category\_Remote\_Play\_on\_Tablet': False,

'category\_VR\_Only': False,

'windows': True,

'mac': False,

'linux': True,

'price': 0, # Free game

'dlc\_count': 0,

}

# Get the prediction

game\_success = predict\_game\_success(user\_input)

print(f"The prediction for your new game: {game\_success}")

### Scraping Code:

import axios from "axios";

import fs, { createReadStream, createWriteStream } from "fs";

import fsPromise from "fs/promises";

import readline from "readline";

import \* as cheerio from "cheerio";

import { parse } from "json2csv";

import csvParser from "csv-parser";

import csvWriteStream from "csv-write-stream";

*const* BASE\_URL = "https://steamspy.com/api.php";

*const* START\_PAGE = 0; // Starting page

*const* MAX\_PAGES = 80; // Adjust based on the expected number of pages (e.g., 0–99 for 100,000 entries).

*const* OUTPUT\_FILE = "steamspy\_data.json";

*const* RATE\_LIMIT\_DELAY = 60 \* 1000; // 1 minute per page for 'all' API

*const* RETRY\_BASE\_DELAY = 10 \* 1000; // Start with 10 seconds delay for retries

// Function to fetch a single page

*const* fetchPage = async (*page*) *=>* {

*const* url = `${BASE\_URL}?request=all&page=${*page*}`;

  console.log(`Fetching page ${*page*}...`);

  try {

*const* response = await axios.get(url, { timeout: 30000 }); // 30 seconds timeout

    return response.data;

  } catch (error) {

    console.error(`Error fetching page ${*page*}: ${error.message}`);

    throw error;

  }

};

// Function with retry logic

*const* fetchWithRetry = async (*page*, *retries* = 5, *delay* = RETRY\_BASE\_DELAY) *=>* {

  for (*let* attempt = 1; attempt <= *retries*; attempt++) {

    try {

      return await fetchPage(*page*);

    } catch (error) {

      if (attempt < *retries*) {

        console.log(

          `Retrying page ${*page*} in ${

*delay* / 1000

          } seconds... (Attempt ${attempt}/${*retries*})`

        );

        await new *Promise*((resolve) *=>* setTimeout(resolve, *delay*));

*delay* += RETRY\_BASE\_DELAY; // Increase delay

      } else {

        console.error(

          `Failed to fetch page ${*page*} after ${*retries*} attempts.`

        );

        throw error;

      }

    }

  }

};

// Main function

*const* fetchAllData = async () *=>* {

*let* dataset = {};

  if (fs.existsSync(OUTPUT\_FILE)) {

*const* existingData = fs.readFileSync(OUTPUT\_FILE, "utf-8");

    dataset = JSON.parse(existingData);

    console.log("Existing data loaded.");

  }

  for (*let* page = START\_PAGE; page < MAX\_PAGES; page++) {

    try {

*const* pageData = await fetchWithRetry(page);

      dataset = { ...dataset, ...pageData }; // Merge current page data into dataset

      fs.writeFileSync(OUTPUT\_FILE, JSON.stringify(dataset, null, 2)); // Save to file after each page

      console.log(`${new *Date*().toLocaleTimeString()} | Page ${page} saved.`);

      await new *Promise*((resolve) *=>* setTimeout(resolve, RATE\_LIMIT\_DELAY)); // Rate limit delay

    } catch (error) {

      console.error(`Skipping page ${page} due to repeated errors.`);

    }

  }

  console.log(`Data fetching completed. Total pages fetched: ${MAX\_PAGES}`);

};

// Function to display the existing dataset

*const* displayDataset = () *=>* {

  if (fs.existsSync(OUTPUT\_FILE)) {

*const* data = fs.readFileSync(OUTPUT\_FILE, "utf8");

    console.log(JSON.parse(data));

  } else {

    console.log("No dataset found. Please scrape data first.");

  }

};

*function* mergeJSONFilesSync(*file1Path*: *string*, *file2Path*: *string*, *outputFilePath*: *string*): *void* {

  try {

    // Read and parse the first file

*const* file1Data = fs.readFileSync(*file1Path*, "utf-8");

*const* json1 = JSON.parse(file1Data);

    // Read and parse the second file

*const* file2Data = fs.readFileSync(*file2Path*, "utf-8");

*const* json2 = JSON.parse(file2Data);

    // Merge the object (assuming they are JSON objects)

*const* mergedData = { ...json1, ...json2 };

    // Write the merged data to the output file

    fs.writeFileSync(

*outputFilePath*,

      JSON.stringify(mergedData, null, 4),

      "utf-8"

    );

    console.log(`Merged data saved to ${*outputFilePath*}`);

  } catch (error) {

    console.error("An error occurred:", error);

  }

}

async *function* getPeakCCU(*appID*: *string*): *Promise*<{ hourly\_peak: *number* | *null*; all\_time\_peak: *number* | *null* }> {

*const* url = `https://steamcharts.com/app/${*appID*}`;

  try {

*const* { data } = await axios.get(url, {

      headers: {

        "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/117.0.0.0 Safari/537.36" } });

    // Load the page content into Cheerio

*const* $ = cheerio.load(data);

    // Extract the 24-hour peak and all-time peak CCU

*const* hourlyPeakText = $("div.app-stat")

      .eq(1)

      .find("span.num")

      .text()

      .trim();

*const* allTimePeakText = $("div.app-stat")

      .eq(2)

      .find("span.num")

      .text()

      .trim();

    // Parse the numbers and remove commas

*const* hourlyPeak = parseInt(hourlyPeakText.replace(/,/g, ""), 10);

*const* allTimePeak = parseInt(allTimePeakText.replace(/,/g, ""), 10);

    if (isNaN(hourlyPeak) || isNaN(allTimePeak)) {

      console.error(`Could not find valid peak CCU for appID ${*appID*}`);

      return { hourly\_peak: null, all\_time\_peak: null };

    }

    return { hourly\_peak: hourlyPeak, all\_time\_peak: allTimePeak };

  } catch (error) {

    console.error(`Error fetching peak CCU for appID ${*appID*}:`, error.message);

    return { hourly\_peak: null, all\_time\_peak: null };

  }

}

// Function to read the JSON file and fetch player counts

async *function* processAppIDs(*jsonFilePath*: *string*, *outputFilePath*: *string*): *Promise*<*void*> {

  try {

*const* PEAK\_CCU\_DELAY = 1000; // Delay between requests in milliseconds

*const* data = JSON.parse(await fsPromise.readFile(*jsonFilePath*, "utf-8"));

*const* appIDs = *Object*.keys(data);

    console.log(`Total appIDs found: ${appIDs.length} | First: ${appIDs[0]} | Last: ${

appIDs[appIDs.length - 1]}`);

*const* START\_THRESHOLD = 0.1;

*const* START\_INDEX = Math.floor(appIDs.length \* START\_THRESHOLD); // Start from the beginning

*const* END\_THRESHOLD = 0.5; // Threshold for splitting the appIDs

*const* END\_INDEX = Math.floor(appIDs.length \* END\_THRESHOLD);

*const* newAppIDs = appIDs.slice(START\_INDEX, END\_INDEX);

*const* totalAppIDs = newAppIDs.length;

    // Prepare results

*let* results: *Record*<*string*, *any*> = {};

*let* count = 0;

*let* actualCount = 0;

*let* skips: *string*[] = [];

    try {

*const* outputFileData = await fsPromise.readFile(*outputFilePath*, "utf-8");

      results = JSON.parse(outputFileData);

      count = *Object*.keys(results).length;

      console.log(`Resuming from previous progress... Previously ${count} appIDs processed.`);

    } catch {

      console.log(`No existing progress found. Starting fresh.`);

      count = 0;

    }

    if (newAppIDs.length === 0) {

      console.log("No appIDs to process in the selected range.");

      return;

    }

    count = 0;

    for (*const* appID of newAppIDs) {

      if (results[appID]) {

        console.log(`Skipping already processed appID ${appID}`);

        count++;

        continue; // Skip already processed appIDs

      }

*const* appName = data[appID]?.name || "Unknown App";

*const* { hourly\_peak, all\_time\_peak } = await getPeakCCU(appID);

      if (hourly\_peak !== null && all\_time\_peak !== null) {

        results[appID] = {

          appid: appID,

          name: appName,

          hourly\_peak\_ccu: hourly\_peak,

          all\_time\_peak\_ccu: all\_time\_peak,

        };

        await fsPromise.writeFile(

*outputFilePath*,

          JSON.stringify(results, null, 4),

          "utf-8"

        );

        actualCount++;

      } else {

        skips.push(appID);

      }

      count++;

      console.log(`Waiting\n`);

      await new *Promise*((resolve) *=>* setTimeout(resolve, PEAK\_CCU\_DELAY));

    }

  } catch (error) {

    console.error("An error occurred:", error.message);

  }

}

async *function* mergeFilesFinal(*playerCountsPath*: *string*, *steamSpyDataPath*: *string*, *outputFilePath*: *string*): *Promise*<*void*> {

  try {

*const* playerCountsData = JSON.parse(await fsPromise.readFile(*playerCountsPath*, "utf-8"));

*const* steamSpyData = JSON.parse( await fsPromise.readFile(*steamSpyDataPath*, "utf-8") );

const combinedData: *Record*<*string*, *any*> = {};

    // Merge data based on appid

    for (*const* appID in steamSpyData) {

      if (playerCountsData[appID]) {

        combinedData[appID] = { ...steamSpyData[appID], ...playerCountsData[appID] };

      } else {

        combinedData[appID] = steamSpyData[appID]; // Include SteamSpy data if no matching player counts

      }

    }

    // Add entries from player counts that are not in SteamSpy

    for (*const* appID in playerCountsData) {

      if (!combinedData[appID]) {

        combinedData[appID] = playerCountsData[appID];

      }

    }

    await fsPromise.writeFile(

*outputFilePath*,

      JSON.stringify(combinedData, null, 4),

      "utf-8"

    );

    console.log(`Merged data successfully written to ${*outputFilePath*}`);

  } catch (error) {

    console.error("An error occurred while merging files:", error.message);

  }

}

async *function* jsonToCsv(*inputFileName*: *string*, *outputFileName*: *string*) {

  try {

    // Read and parse the JSON file

*const* rawData = JSON.parse(

      await fsPromise.readFile(*inputFileName*, "utf-8")

    );

    if (typeof rawData !== "object" || *Array*.isArray(rawData)) {

      throw new *Error*("Input JSON must be an object where values are objects.");

    }

    // Flatten the JSON object to an array of records

*const* records = *Object*.values(rawData);

    // Convert JSON records to CSV using json2csv

*const* csvData = parse(records);

    // Write CSV data to the output file

    await fsPromise.writeFile(*outputFileName*, csvData, "utf-8");

    console.log(`CSV file has been saved as "${*outputFileName*}".`);

  } catch (error) {

    console.error("Error:", error.message);

  }

}

*const* removeDuplicates = async (*inputFile*, *outputFile*) *=>* {

*const* appidSet = new *Set*();

*const* uniqueRows = [];

  // Read CSV and collect unique rows

*const* readCSV = async () *=>* {

    return new *Promise*((resolve, reject) *=>* {

*const* readStream = createReadStream(*inputFile*);

      readStream

        .pipe(csvParser())

        .on("data", (*row*) *=>* {

*const* appid = *row*.appid;

          if (appid && !appidSet.has(appid)) {

            appidSet.add(appid);

            uniqueRows.push(*row*);

          }

        })

        .on("end", resolve)

        .on("error", reject);

    });

  };

  await readCSV();

  // Write unique rows to a new CSV file

*const* writeCSV = async () *=>* {

    return new *Promise*((resolve, reject) *=>* {

*const* writer = csvWriteStream({ headers: *Object*.keys(uniqueRows[0]) });

*const* writeStream = createWriteStream(*outputFile*);

      writer.pipe(writeStream);

      uniqueRows.forEach((*row*) *=>* writer.write(*row*));

      writer.end();

      writeStream.on("finish", resolve);

      writeStream.on("error", reject);

    });

  };

  await writeCSV();

  console.log(`Cleaned data without duplicates written to ${*outputFile*}`);

};

*const* combineFiles = async (*fileSteamDB*, *fileMain*, *outputFile*) *=>* {

*const* fileAData = {};

*const* fileBData = [];

  // Read data from File A

*const* readFileA = () *=>*

    new *Promise*((resolve, reject) *=>* {

      fs.createReadStream(*fileSteamDB*)

        .pipe(csvParser())

        .on("data", (*row*) *=>* {

          // Store File A data by appid for easy lookup

          fileAData[*row*.appid] = {

            supported\_os: *row*.supported\_os,

            technologies: *row*.technologies,

          };

        })

        .on("end", resolve)

        .on("error", reject);

    });

  // Read data from File B

*const* readFileB = () *=>*

    new *Promise*((resolve, reject) *=>* {

      fs.createReadStream(*fileMain*)

        .pipe(csvParser())

        .on("data", (*row*) *=>* {

          fileBData.push(*row*);

        })

        .on("end", resolve)

        .on("error", reject);

    });

  await *Promise*.all([readFileA(), readFileB()]);

  // Combine data from File A and File B

*const* combinedData = fileBData.map((*row*) *=>* {

*const* appId = *row*.appid;

    return {

      ...*row*, // Include all fields from File B

      supported\_os: fileAData[appId]?.supported\_os || "N/A", // Add or fallback to 'N/A'

      technologies: fileAData[appId]?.technologies || "N/A", // Add or fallback to 'N/A'

    };

  });

  // Write combined data to a new CSV file

*const* writer = csvWriteStream({

    headers: [

      "appid",

      "name",

      "developer",

      "publisher",

      "positive",

      "negative",

      "userscore",

      "owners",

      "price",

      "initialprice",

      "discount",

      "hourly\_peak\_ccu",

      "all\_time\_peak\_ccu",

      "supported\_os", "technologies",

    ],

  });

  writer.pipe(fs.createWriteStream(*outputFile*));

  combinedData.forEach((*row*) *=>* writer.write(*row*));

  writer.end();

  console.log(`Combined data written to ${*outputFile*}`);

};

*const* mainMenu = () *=>* {

*const* rl = readline.createInterface({

    input: process.stdin,

    output: process.stdout,

  });

*const* showMenu = () *=>* {

    rl.question("Enter your choice: ", async (*choice*) *=>* {

*const* sanitisedChoice = *choice*.trim().toLowerCase();

      if (sanitisedChoice === "1") {

        console.log("Starting data scrape...");

        // Start the process

        try {

          await fetchAllData();

        } catch (error) {

          console.error(`Script failed: ${error.message}`);

        }

        console.log("Data scrape completed.");

        showMenu();

      } else if (sanitisedChoice === "2") {

        console.log("Displaying existing data:");

        displayDataset();

        showMenu();

      } else if (sanitisedChoice === "3") {

        console.log("Merging JSON files...");

*const* file1 = "hamza\_player\_count.json"; // "steamspy\_data\_50.json";

*const* file2 = "player\_counts.json"; // "steamspy\_data.json";

*const* outputFile = "final\_player\_counts.json";

        mergeJSONFilesSync(file1, file2, outputFile);

        console.log("JSON files merged successfully.");

        showMenu();

      } else if (sanitisedChoice === "4") {

        console.log("Scraping SteamDB Data...");

*const* jsonFilePath = "steamspy\_data\_combined.json"; // Input JSON file path

*const* outputFilePath = "player\_counts.json"; // Output JSON file path

*const* apiKey = "YOUR\_STEAM\_API\_KEY"; // Replace with your Steam API key

        await processAppIDs(jsonFilePath, outputFilePath, apiKey);

        showMenu();

      } else if (sanitisedChoice === "5") {

        // Call the function to merge files

        await mergeFilesFinal(

          "final\_player\_counts.json",

          "final\_steamspy\_data.json",

          "final\_combined.json"

        );

        await jsonToCsv("final\_combined.json", "final\_output.csv");

        showMenu();

      } else if (sanitisedChoice === "6") {

        await removeDuplicates("steamdb\_games\_data.csv", "steamdb\_games\_data\_clean.csv");

        showMenu();

      } else if (sanitisedChoice === "7") {

        await combineFiles( "cloned/steam-scrape/data/5\_steamdb.csv", "cloned/steam-scrape/data/4\_output.csv", "6\_final.csv");

        showMenu();

      } else if (sanitisedChoice === "q") {

        console.log("Exiting program.");

        rl.close();

      } else {

        console.log("Invalid choice. Please try again.");

        showMenu();

      }

    });

  };

  showMenu();

};

mainMenu();

### Data Cleaning Code

import pandas as pd

file\_path = 'result.csv'

data = pd.read\_csv('result.csv')

columns\_to\_fill\_false = [

"Action", "Free to Play", "Adventure", "Massively Multiplayer", "Strategy",

"Indie", "RPG", "Casual", "Simulation", "Early Access", "Sports", "Racing",

"Violent", "Gore", "Movie", "Animation & Modeling", "Design & Illustration",

"Education", "Software Training", "Utilities", "Web Publishing",

"Sexual Content", "Nudity", "Audio Production"

]

data[columns\_to\_fill\_false] = data[columns\_to\_fill\_false].fillna(False)

def calculate\_midpoint(range\_str):

try:

min\_val, max\_val = map(int, range\_str.split('-'))

return (min\_val + max\_val) // 2

except:

return None # Return None if parsing fails

data["owners"] = data["owners"].apply(calculate\_midpoint)

categories\_split = data['categories'].str.strip("[]").str.replace("'", "").str.split(", ")

all\_categories = set()

for cat\_list in categories\_split.dropna():

all\_categories.update(cat\_list)

for category in all\_categories:

column\_name = f"category\_{category.replace(' ', '\_')}"

data[column\_name] = categories\_split.apply(lambda x: category in x if isinstance(x, list) else False)

data = data.drop(columns=['categories'], errors='ignore')

data = data.drop(columns=['min\_owners', 'max\_owners'], errors='ignore')

missing\_counts = data.isnull().sum()

print("Missing values per column:")

print(missing\_counts)

output\_path = 'cleaned\_result.csv'

data.to\_csv('cleaned\_data.csv', index=False)

print(f"Cleaned dataset saved to {output\_path}")

### Findings:

This structured analysis provides a comprehensive view of the gaming landscape on Steam and empowers stakeholders to make data-driven decisions, showing that:

* Games with action elements remain the most popular games on the market, with examples of PUBG, Counter Strike, Dota and Elden Ring as proof.
* Steam, although also a distributor of other games, the top games remain to be of Valve, which is Steam’s parent company.
* Nearly 67% of all the games released on the Steam platform support Windows platform, which confirms that Microsoft’s platform remains at the core of the PC Gaming industry.
* Majority of the games on the Steam platform are paid, with the average price being roughly $11 US Dollars.
* Call of Duty and FIFA games remain to be the most expensive games on the market year over year.
* Players are purchasing more indie games (games that are produced by individuals or small teams that are not financially backed by major studios).
* The best genre to target is Action, where the minimum price is $6 US Dollars, indicating that (indie) developers can price their games at this price point without sacrificing sales and earn higher revenue.

**Conclusion:**

This report presents the systematic process of collecting, cleaning, and analyzing game data, with a focus on identifying key trends that can benefit both players and developers in understanding the market better. While the project provides valuable insights, certain limitations remain. For example, adding additional features such as game engine and technology data could offer a deeper understanding of the tools used by developers today. Incorporating time/history-based data could also enhance machine learning models and provide more nuanced insights. Additionally, including user reviews and granular gameplay preferences would further refine recommendations.

During the development, we encountered significant challenges, particularly with data scraping. APIs often lack the necessary information, and strict rate limiting and web scraping policies from some sources made data collection more difficult than anticipated.

Future work could focus on expanding the data sources and exploring more advanced recommendation techniques, such as collaborative filtering, to improve the accuracy of suggestions. Overall, this project highlights the potential of data science to create personalized experiences and emphasizes the need for continuous refinement in model development.