Predicting the Success of Indie Games on Steam Using Metadata and Machine Learning Models

Derek W. Graves

Northwest Missouri State University, Maryville MO 64468, USA
`S573443@nwmissouri.edu` and `derek.graves4@outlook.com`

Abstract. This study aims to predict the commercial success of indie games on the Steam platform by analyzing game metadata. Using data collected from the Steam API, machine learning techniques, including Random Forest and Logistic Regression, were employed to identify key attributes driving game popularity. By examining factors such as gameplay features, pricing strategies, and developer information, this research provides actionable insights for indie developers seeking to enhance player engagement and maximize market success. Exploratory data analysis, featuring distribution plots, boxplots, and correlation matrices, was conducted to uncover significant relationships between game features and popularity metrics.

Keywords: machine learning · data analytics · Steam · indie games

1 Introduction

The rapid growth of the indie game industry on platforms like Steam has created an unprecedented opportunity for developers to reach global audiences. However, with thousands of games released each year, understanding the factors that contribute to a game's success is critical for indie developers looking to stand out. This research aims to analyze and predict the success potential of indie games on Steam by leveraging available metadata.

To achieve this, data was collected using the Steam Web API and analyzed using machine learning models, including Random Forest and Logistic Regression. These models were chosen for their ability to handle complex datasets and provide interpretable insights into the attributes most associated with game popularity. Exploratory data analysis (EDA) techniques, such as distribution plots, boxplots, and correlation matrices, were employed to uncover initial patterns and inform feature selection for the models.

Previous studies have demonstrated the efficacy of machine learning in analyzing complex datasets, particularly in predicting outcomes based on metadata [1]. Similarly, research on game success prediction underscores the importance of identifying relevant features to achieve accurate predictions [2]. This study builds upon these works by providing practical insights for indie developers to enhance player engagement and optimize their games' market success.

The ultimate goal of this research is to empower developers with data-driven strategies, enabling them to navigate the competitive landscape of the indie game market effectively.

1.1 Research Goals

The primary goals of this study are as follows:

- To identify which game metadata features, such as price, recommendations, genres, and developer reputation, are significantly correlated with the success of an indie game [2].
- To apply predictive machine learning techniques, including Random Forest and Logistic Regression, to forecast game success based on the identified features.
- To provide actionable insights that indie game developers can leverage to enhance player engagement, optimize game design, and improve market reception.

2 Data Collection

The dataset for this study was collected using the Steam Web API, which offers comprehensive metadata for all games on the platform, including gameplay features, pricing, developer details, and user engagement metrics such as user recommendations. Documentation from the Steam API Documentation was referenced to understand the API's various parameters and response formats.

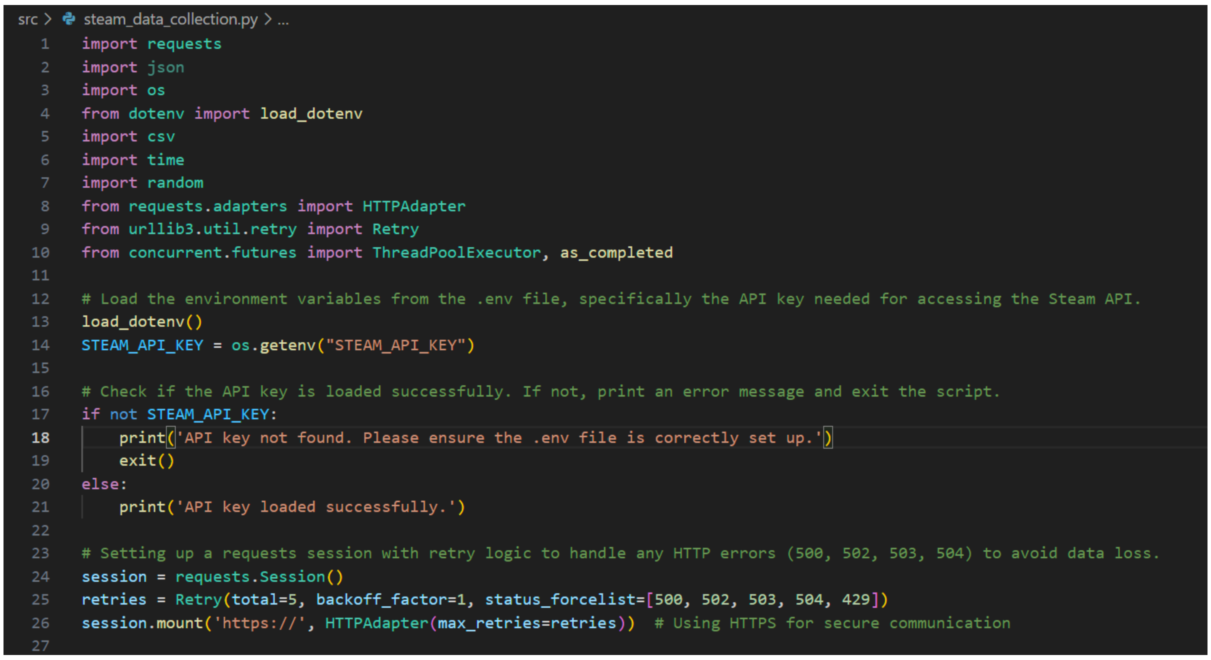
2.1 Source of Data

The dataset consists of data from indie games released between 2010 and 2024. Data was gathered from various regions, including North America, Europe, and Asia, ensuring a diverse representation of games across different market segments. This broad dataset captures a range of game genres such as action, role-playing, puzzle, and adventure, allowing the analysis to explore how genre influences game success.

2.2 Data Extraction Procedure

To collect data from the Steam Web API, several key steps were undertaken, including setting up the API environment, fetching detailed game data, and ensuring the dataset was balanced for analysis.

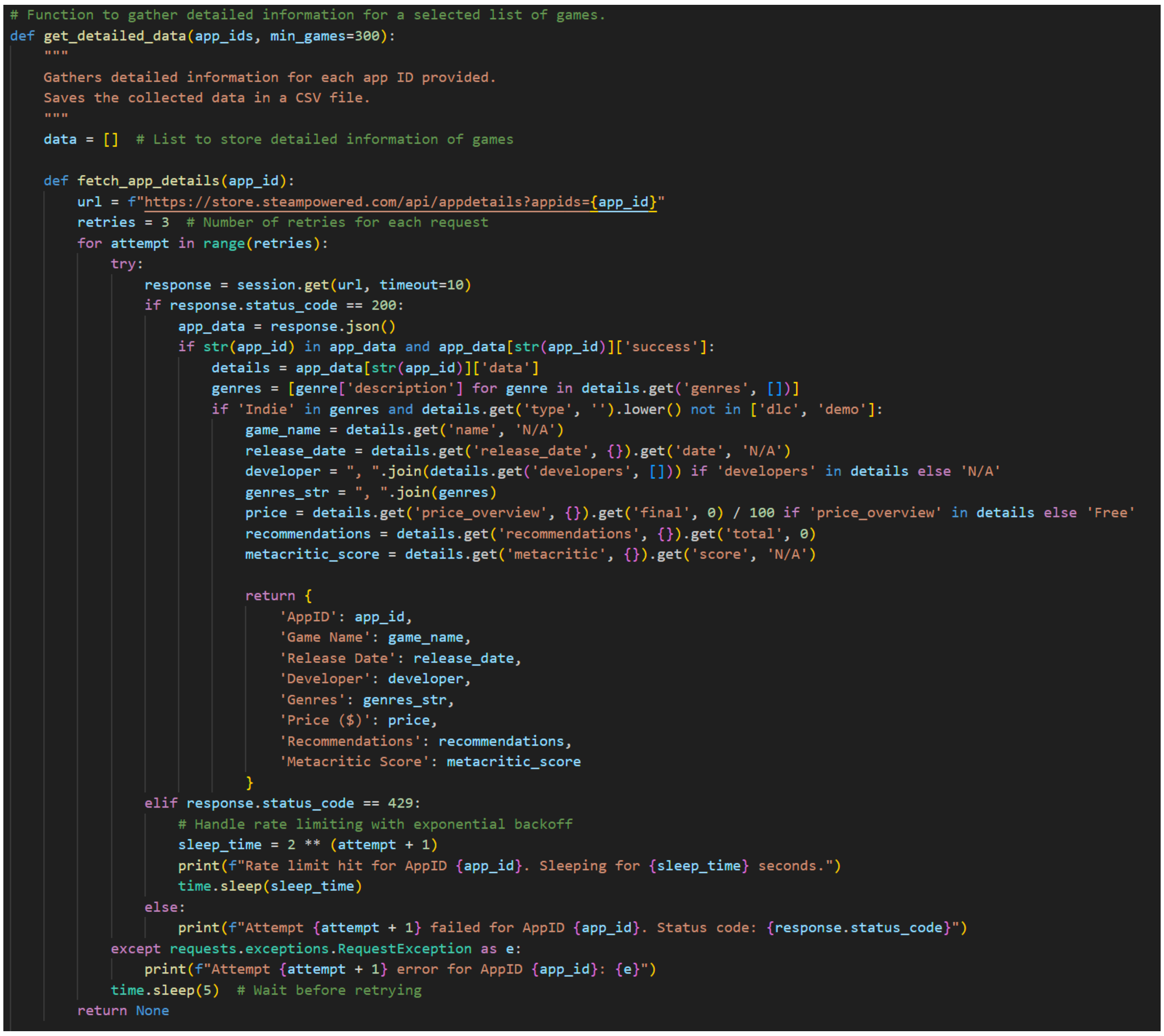
API Setup Environment The Steam Web API was used to collect the required game metadata. Figure 1 shows the Python script used to set up the API environment, providing access to data such as game names, release dates, and game statistics.



```python
import requests
import json
import os
from dotenv import load_dotenv
import csv
import time
import random
from requests.adapters import HTTPAdapter
from urllib3.util.retry import Retry
from concurrent.futures import ThreadPoolExecutor, as_completed

# Load the environment variables from the .env file, specifically the API key needed for accessing the Steam API.
load_dotenv()
STEAM_API_KEY = os.getenv("STEAM_API_KEY")

# Check if the API key is loaded successfully. If not, print an error message and exit the script.
if not STEAM_API_KEY:
    print('API key not found. Please ensure the .env file is correctly set up.')
    exit()
else:
    print('API key loaded successfully.')

# Setting up a requests session with retry logic to handle any HTTP errors (500, 502, 503, 504) to avoid data loss.
session = requests.Session()
retries = Retry(total=5, backoff_factor=1, status_forcelist=[500, 502, 503, 504, 429])
session.mount('https://', HTTPAdapter(max_retries=retries))  # Using HTTPS for secure communication
```

Fig. 1: Setting up the API environment for data collection from the Steam Web API.

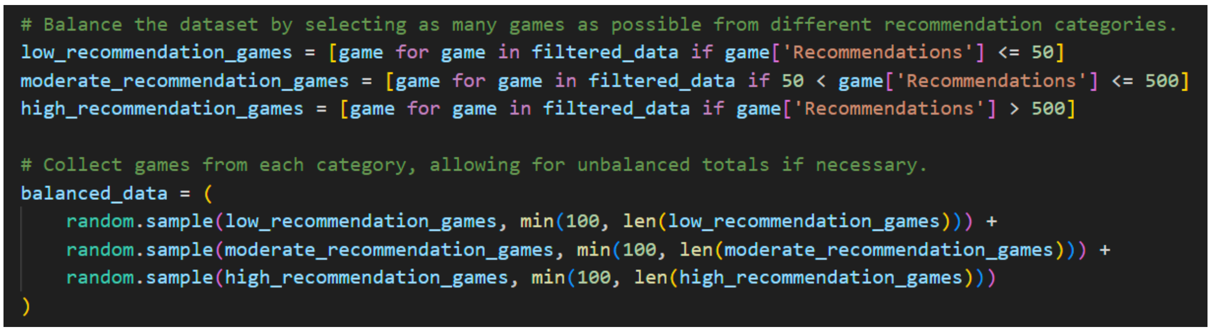
Fetching Detailed Game Data To obtain detailed metadata, multiple API calls were made using the Requests library in Python, with provisions for handling rate limits and retries to manage transient errors. Figure 2 illustrates how the game data was fetched, including a retry mechanism to manage server timeouts.



```python
# Function to gather detailed information for a selected list of games.
def get_detailed_data(app_ids, min_games=300):
    """
    Gathers detailed information for each app ID provided.
    Saves the collected data in a CSV file.
    """
    data = []  # List to store detailed information of games

    def fetch_app_details(app_id):
        url = f"https://store.steampowered.com/api/appdetails?appids={app_id}"
        retries = 3  # Number of retries for each request
        for attempt in range(retries):
            try:
                response = session.get(url, timeout=10)
                if response.status_code == 200:
                    app_data = response.json()
                    if str(app_id) in app_data and app_data[str(app_id)]['success']:
                        details = app_data[str(app_id)]['data']
                        genres = [genre['description'] for genre in details.get('genres', [])]
                        if 'Indie' in genres and details.get('type', '').lower() not in ['dlc', 'demo']:
                            game_name = details.get('name', 'N/A')
                            release_date = details.get('release_date', {}).get('date', 'N/A')
                            developer = ", ".join(details.get('developers', [])) if 'developers' in details else 'N/A'
                            genres_str = ", ".join(genres)
                            price = details.get('price_overview', {}).get('final', 0) / 100 if 'price_overview' in details else 'Free'
                            recommendations = details.get('recommendations', {}).get('total', 0)
                            metacritic_score = details.get('metacritic', {}).get('score', 'N/A')

                            return {
                                'AppID': app_id,
                                'Game Name': game_name,
                                'Release Date': release_date,
                                'Developer': developer,
                                'Genres': genres_str,
                                'Price ($)': price,
                                'Recommendations': recommendations,
                                'Metacritic Score': metacritic_score
                            }
                elif response.status_code == 429:
                    # Handle rate limiting with exponential backoff
                    sleep_time = 2 ** (attempt + 1)
                    print(f"Rate limit hit for AppID {app_id}. Sleeping for {sleep_time} seconds.")
                    time.sleep(sleep_time)
                else:
                    print(f"Attempt {attempt + 1} failed for AppID {app_id}. Status code: {response.status_code}")
            except requests.exceptions.RequestException as e:
                print(f"Attempt {attempt + 1} error for AppID {app_id}: {e}")
            time.sleep(5)  # Wait before retrying
        return None
```

Fig. 2: Python code snippet for fetching detailed game data from the Steam Web API.

Balanced Data Selection To create a dataset that fairly represented different levels of game popularity, games were selected across various popularity levels (low, moderate, and high). Figure 3 shows the code used to balance the dataset to reduce bias and ensure that different types of games were adequately represented.



```python
# Balance the dataset by selecting as many games as possible from different recommendation categories.
low_recommendation_games = [game for game in filtered_data if game['Recommendations'] <= 50]
moderate_recommendation_games = [game for game in filtered_data if 50 < game['Recommendations'] <= 500]
high_recommendation_games = [game for game in filtered_data if game['Recommendations'] > 500]

# Collect games from each category, allowing for unbalanced totals if necessary.
balanced_data = (
    random.sample(low_recommendation_games, min(100, len(low_recommendation_games))) +
    random.sample(moderate_recommendation_games, min(100, len(moderate_recommendation_games))) +
    random.sample(high_recommendation_games, min(100, len(high_recommendation_games)))
)
```

Fig. 3: Balancing the dataset by selecting games with different levels of popularity.

2.3 Key Considerations and Challenges

While the Steam Web API provided detailed and structured data, several challenges and considerations were addressed during the data collection process:

– **Nature of API Data:** Since the API reflects real-time updates, the dataset represents a snapshot of available data at the time of collection, leading to potential variations if collected at another time.
– **Rate Limiting and Retries:** API rate limits were managed using pauses and retry logic, as illustrated in Figure 2.
– **Balanced Dataset:** The dataset was balanced to reduce popularity bias, ensuring a more comprehensive analysis, as shown in Figure 3.

The structured approach to data collection ensured a rich and well-balanced dataset, suitable for understanding the factors contributing to the success of indie games on Steam.

2.4 Source of Data

The dataset consists of data from indie games released between 2010 and 2024. Data was gathered from various regions, including North America, Europe, and Asia, ensuring a diverse representation of games across different market segments. This broad dataset captures a range of game genres such as action, role-playing, puzzle, and adventure, allowing the analysis to explore how genre influences game success.

2.5 Data Extraction Procedure

The dataset was extracted using Python, leveraging the `Requests` library to interact with the Steam Web API. The following steps were undertaken to ensure high-quality and relevant data collection:

– **API Integration:** Metadata was retrieved by calling the Steam Web API through the `Requests` library in Python. The API provided detailed information for each game, including price, release date, developer, genres, and the number of user recommendations.
– **Rate Limiting and Retry Logic:** To avoid exceeding API limits, pauses were implemented after every 10 requests. A retry mechanism using the `Retry` feature from the `urllib3` library ensured resilience against transient errors, such as server timeouts, and maintained robust data collection.
– **Multithreading for Efficiency:** To handle the large volume of games efficiently, multithreading was implemented. This allowed multiple concurrent requests to the Steam API while adhering to rate-limiting constraints, significantly reducing the overall data collection time.
– **Data Filtering:** The dataset was filtered to focus exclusively on completed indie games, ensuring data relevance for the analysis. Games were excluded if they met any of the following criteria:
  • Adult content
  • Demos and downloadable content (DLC)
  • Games still in early access
– **Balanced Data Selection:** To minimize bias, the dataset was balanced by including games across varying levels of popularity, categorized based on the number of user recommendations (low, moderate, and high).
– **Data Storage:** The extracted data was saved in CSV format, which included a comprehensive raw dataset as well as a balanced subset tailored for further analysis. This storage approach facilitated efficient access and preprocessing in subsequent stages.

2.6 Key Considerations and Challenges

While the Steam Web API provided detailed and structured data, several challenges and considerations were addressed during the data collection process:

– **The Nature of API Data:** As the API reflects real-time updates, the dataset represents a snapshot of available data at the time of collection. This could lead to variations if the process were repeated at a different time, even with the same request format.
– **Missing Metadata:** Some attributes, such as Metacritic scores, were incomplete or inconsistent across games. These missing values were handled during the data cleaning phase.
– **Variety of Features:** The dataset included a wide range of features, from game pricing to user engagement metrics. This diversity allowed for a robust analysis but required careful preprocessing to ensure uniformity and relevance of features for use in machine learning.
– **Bias in Popularity Metrics:** Games with higher exposure or longer availability on the platform naturally accumulated more recommendations, potentially introducing bias in the dataset. The balancing strategy that was used in requests aimed helped mitigate this issue.

3 Data Cleaning and Exploratory Data Analysis (EDA)

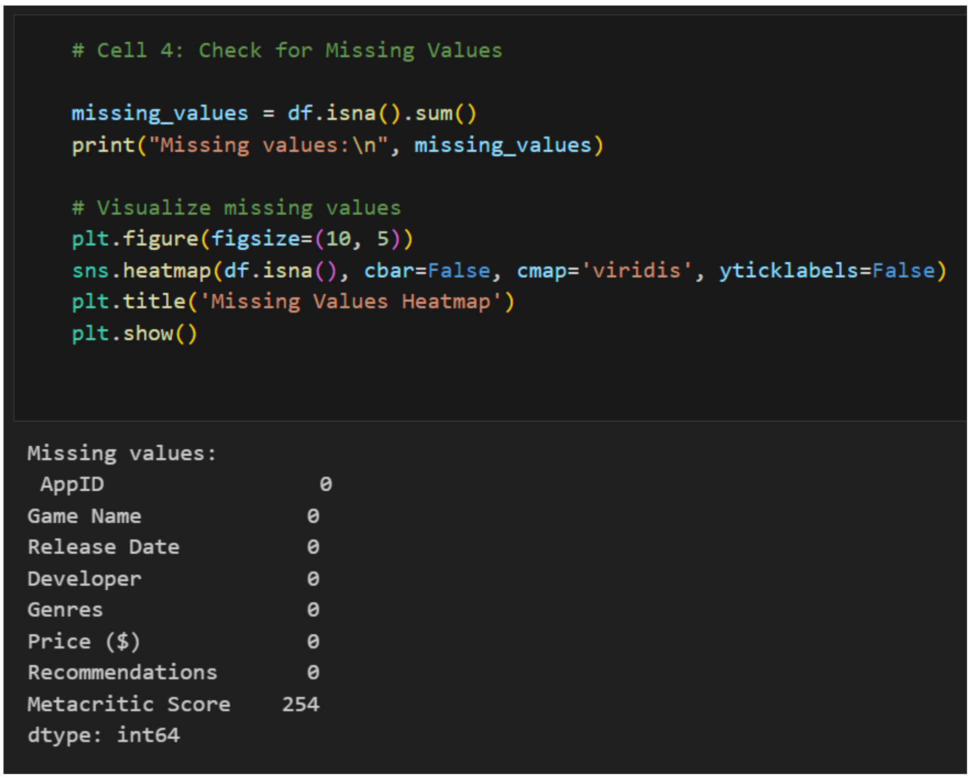
The data cleaning and EDA phases involved filtering the raw dataset, handling missing values, visualizing feature distributions, and exploring relationships between features.

3.1 Data Cleaning Procedure

The raw dataset underwent a structured cleaning process to ensure its quality and suitability for analysis. Key steps in this process included handling missing values, engineering relevant features, normalizing data for modeling, and converting data types for better usability.

Key Cleaning Steps

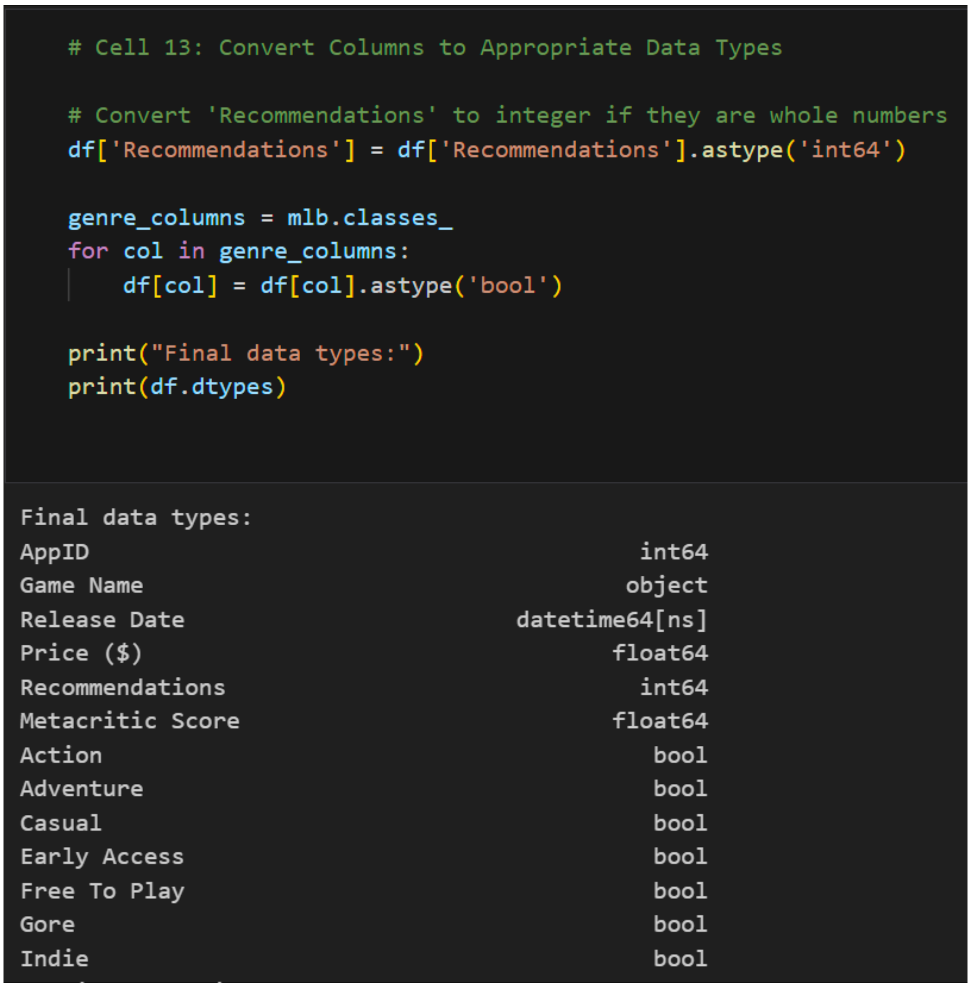
- **Identifying Missing Values:** As depicted in Figure 4, a heatmap was used to identify columns with missing values. This visual helped pinpoint where values were missing, enabling us to make informed decisions on how to handle these gaps.
- **Handling Missing Values:** As shown in Figure 5, missing values in critical fields such as `Game Name` and `Release Date` were excluded. For numerical fields like `Price ($)`, missing values were imputed using the median of the column to minimize bias.
- **Converting Data Types:** Certain columns were converted to the appropriate data types for analysis, as illustrated in Figure 5. For instance, numerical features were converted from strings, and categorical fields were transformed for easier analysis.
- **Feature Engineering:** In Figure 6, the new feature `Years Since Release` was calculated. This feature was created by converting `Release Date` into a numerical format and subtracting the year from 2024 to provide a temporal context for each game. Figure 7 shows the one-hot encoding process used for categorizing game genres.
- **Normalization:** To prepare for machine learning, the `Price ($)` column was scaled using Min-Max normalization, as illustrated in Figure 8. This ensured consistent value ranges across features, which is critical for effective model training.



```python
# Cell 4: Check for Missing Values

missing_values = df.isna().sum()
print("Missing values:\n", missing_values)

# Visualize missing values
plt.figure(figsize=(10, 5))
sns.heatmap(df.isna(), cbar=False, cmap='viridis', yticklabels=False)
plt.title('Missing Values Heatmap')
plt.show()
```

```
Missing values:
 AppID                0
Game Name            0
Release Date         0
Developer            0
Genres               0
Price ($)            0
Recommendations      0
Metacritic Score   254
dtype: int64
```

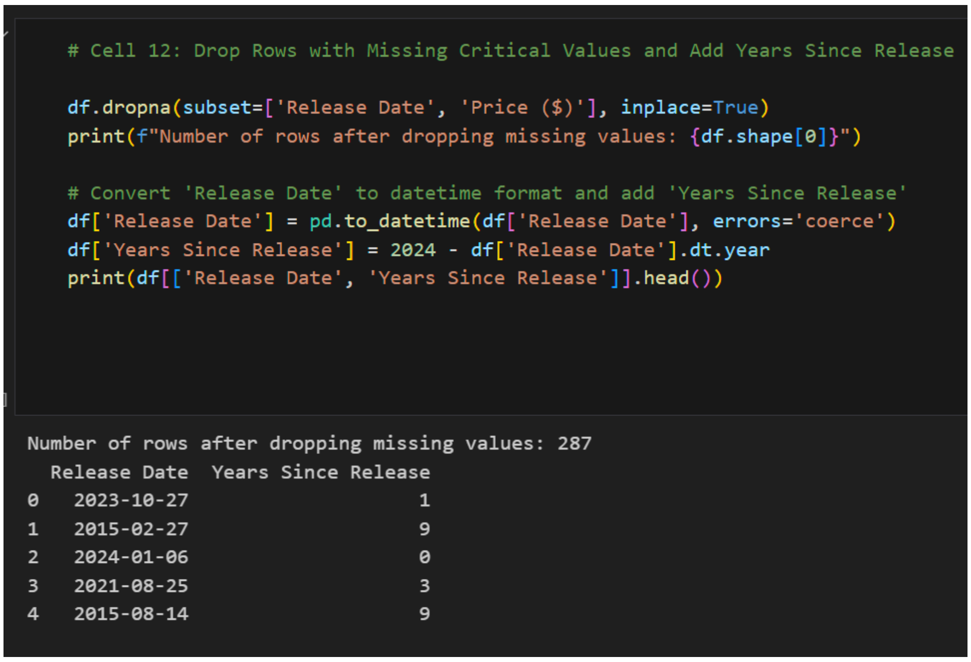
Fig. 4: Identifying missing values using a heatmap to assess the data quality.



```
# Cell 13: Convert Columns to Appropriate Data Types

# Convert 'Recommendations' to integer if they are whole numbers
df['Recommendations'] = df['Recommendations'].astype('int64')

genre_columns = mlb.classes_
for col in genre_columns:
    df[col] = df[col].astype('bool')

print("Final data types:")
print(df.dtypes)
```

```
Final data types:
AppID                                        int64
Game Name                                   object
Release Date                        datetime64[ns]
Price ($)                                  float64
Recommendations                              int64
Metacritic Score                           float64
Action                                        bool
Adventure                                     bool
Casual                                        bool
Early Access                                  bool
Free To Play                                  bool
Gore                                          bool
Indie                                         bool
```

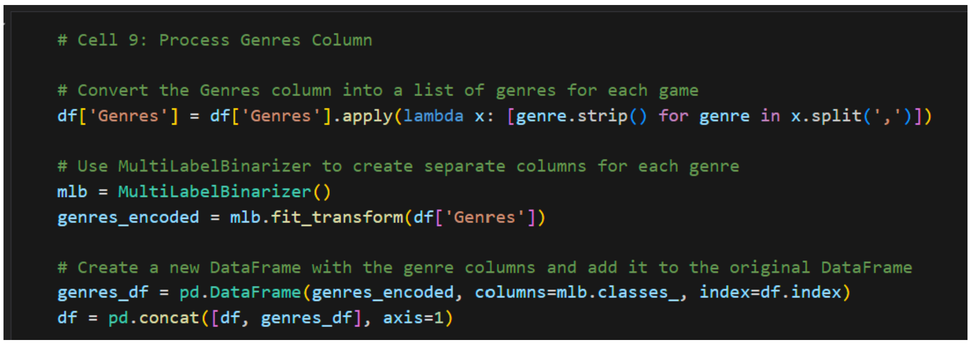
Fig. 5: Handling missing values and engineering new features such as `Years Since Release`.



```
# Cell 12: Drop Rows with Missing Critical Values and Add Years Since Release

df.dropna(subset=['Release Date', 'Price ($)'], inplace=True)
print(f"Number of rows after dropping missing values: {df.shape[0]}")

# Convert 'Release Date' to datetime format and add 'Years Since Release'
df['Release Date'] = pd.to_datetime(df['Release Date'], errors='coerce')
df['Years Since Release'] = 2024 - df['Release Date'].dt.year
print(df[['Release Date', 'Years Since Release']].head())
```

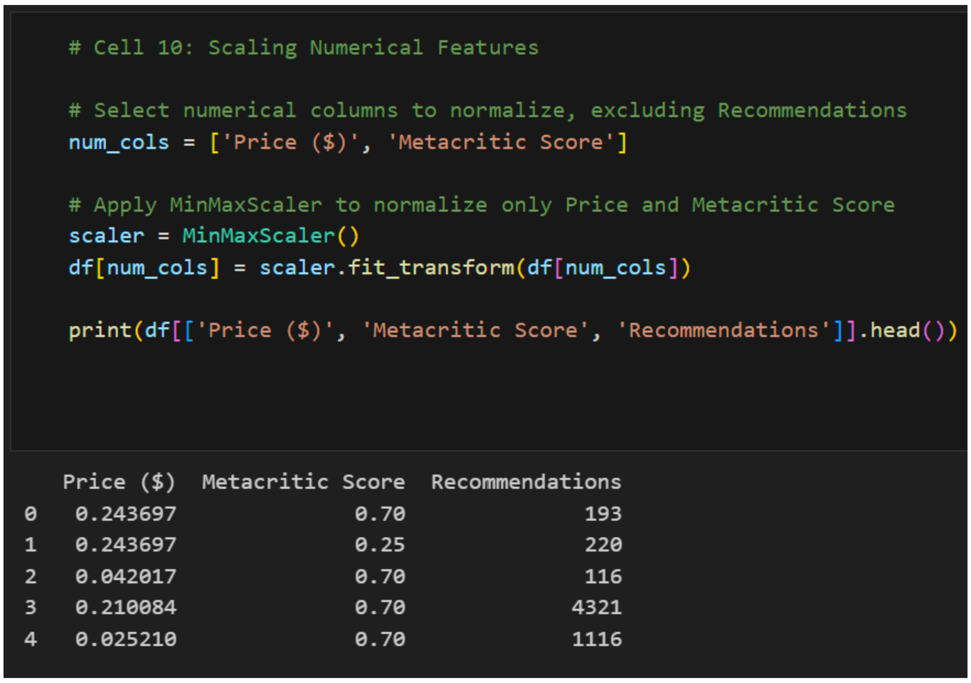
```
Number of rows after dropping missing values: 287
  Release Date  Years Since Release
0   2023-10-27                    1
1   2015-02-27                    9
2   2024-01-06                    0
3   2021-08-25                    3
4   2015-08-14                    9
```

Fig. 6: Feature engineering for `Years Since Release` by transforming the `Release Date`.



```
# Cell 9: Process Genres Column

# Convert the Genres column into a list of genres for each game
df['Genres'] = df['Genres'].apply(lambda x: [genre.strip() for genre in x.split(',')])

# Use MultiLabelBinarizer to create separate columns for each genre
mlb = MultiLabelBinarizer()
genres_encoded = mlb.fit_transform(df['Genres'])

# Create a new DataFrame with the genre columns and add it to the original DataFrame
genres_df = pd.DataFrame(genres_encoded, columns=mlb.classes_, index=df.index)
df = pd.concat([df, genres_df], axis=1)
```

Fig. 7: One-hot encoding game genres for improved categorical representation.



```
# Cell 10: Scaling Numerical Features

# Select numerical columns to normalize, excluding Recommendations
num_cols = ['Price ($)', 'Metacritic Score']

# Apply MinMaxScaler to normalize only Price and Metacritic Score
scaler = MinMaxScaler()
df[num_cols] = scaler.fit_transform(df[num_cols])

print(df[['Price ($)', 'Metacritic Score', 'Recommendations']].head())
```

```
   Price ($)  Metacritic Score  Recommendations
0   0.243697              0.70              193
1   0.243697              0.25              220
2   0.042017              0.70              116
3   0.210084              0.70             4321
4   0.025210              0.70             1116
```

Fig. 8: Scaling numeric features using Min-Max normalization.

The structured cleaning process resulted in a dataset with consistent formatting and robust features, making it ready for analysis and model development.

3.2 Exploratory Data Analysis (EDA)

The EDA phase provided insights into the structure and characteristics of the dataset, aiding in feature selection and hypothesis generation. Visualizations included distribution plots, boxplots, scatter plots, bar charts, and correlation heatmaps.

Distribution Plots The distribution of key numerical features such as price was visualized using histograms.

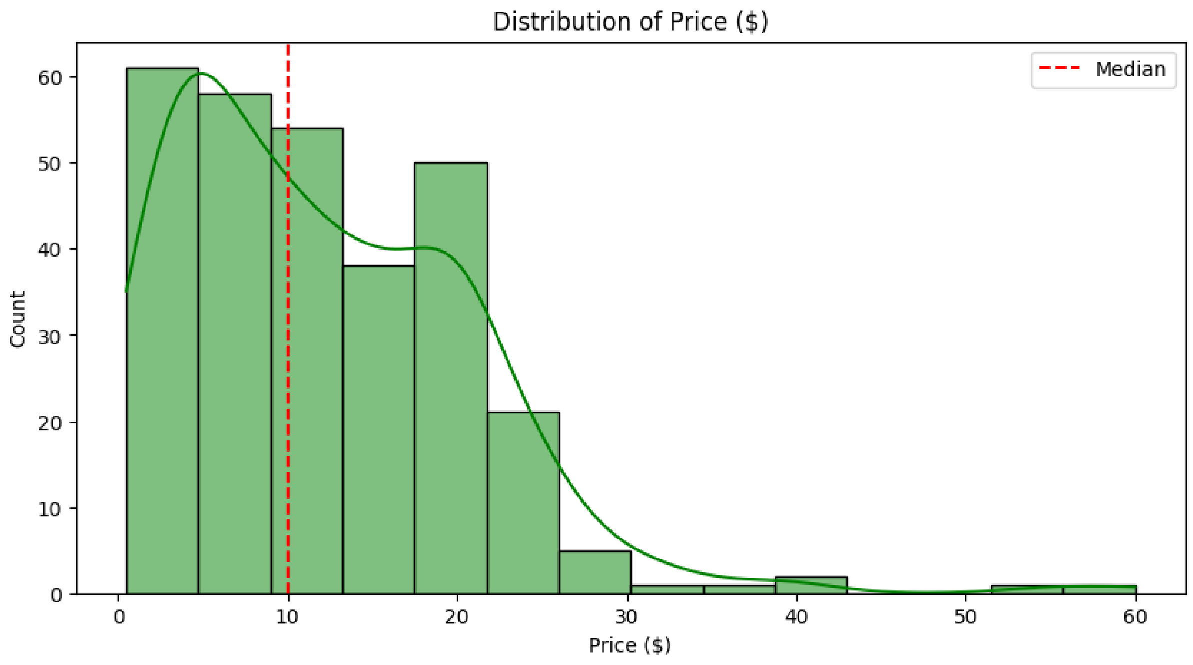


Fig. 9: Distribution of Game Prices. The distribution reveals that most indie games are priced below $20, with some outliers at higher price points. This provides insight into common pricing strategies among indie developers.

Correlation Analysis A heatmap was generated to visualize correlations between numerical features. This analysis was instrumental in understanding relationships among attributes and informing feature selection for model training.

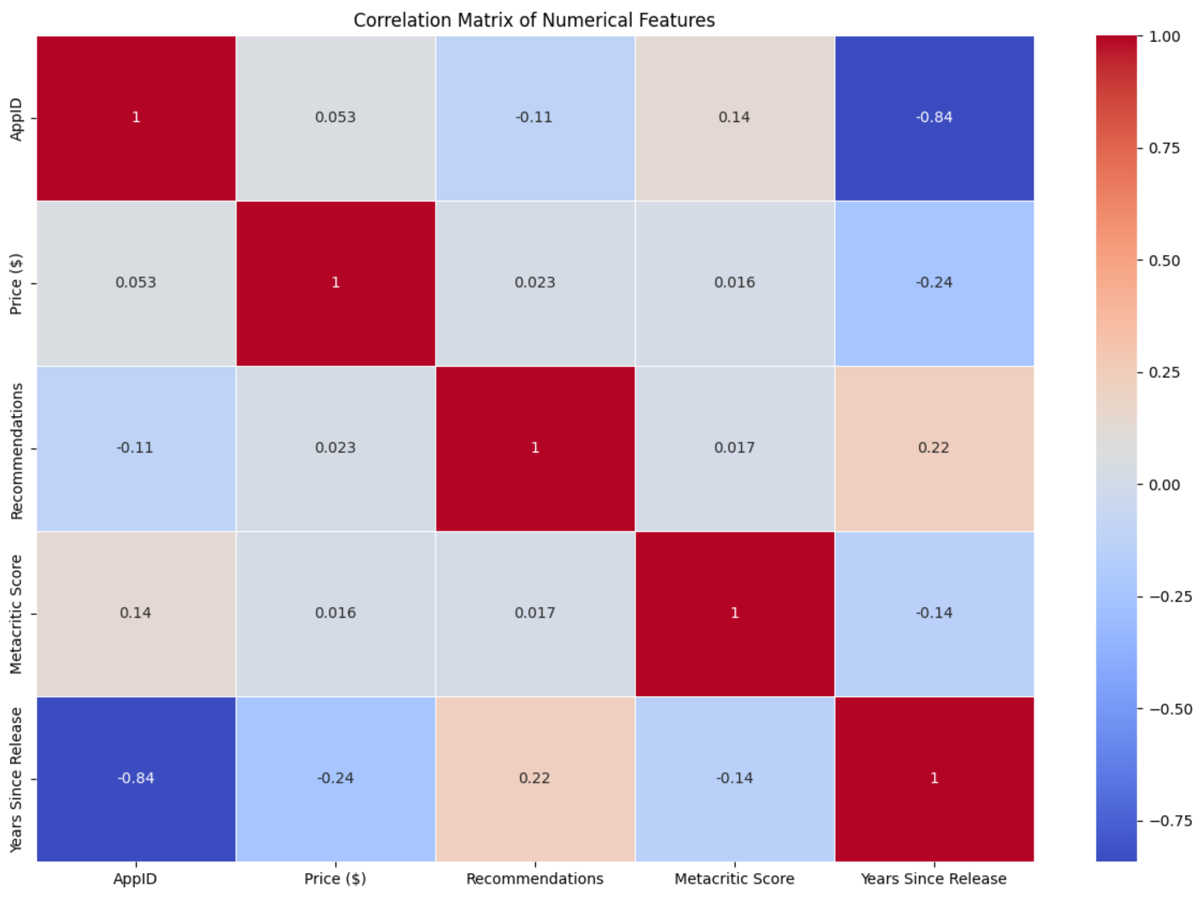


Fig. 10: Correlation Matrix of Numerical Features. The heatmap reveals that most correlations between features are weak, with a notable strong negative correlation between 'Years Since Release' and 'AppID.' It also shows that price and recommendations have a near-zero correlation, indicating that price alone does not significantly affect a game's number of recommendations. This information was crucial in feature selection and understanding the data's overall relationships before modeling

Genre Distributions Heatmap The distribution of different genres was visualized using a heatmap to better understand the presence of genres in the dataset.

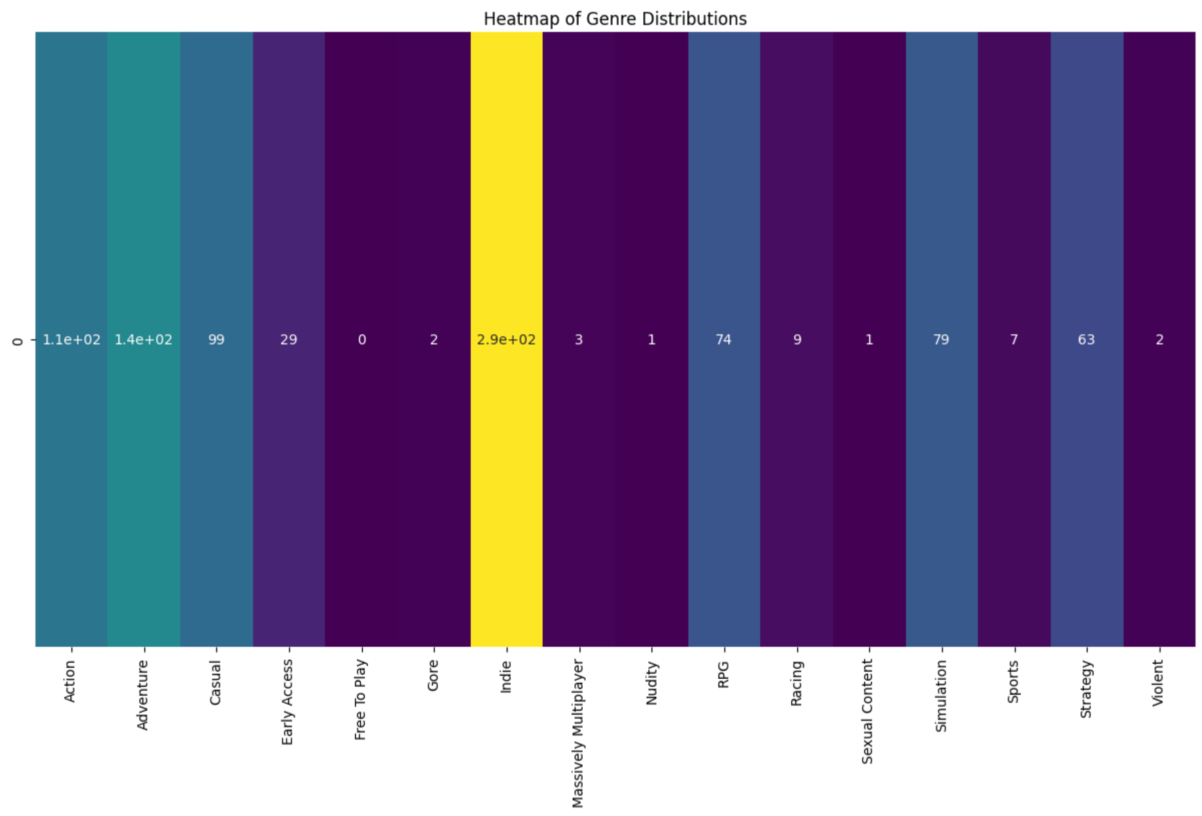


Fig. 11: Heatmap of Genre Distributions. This plot helps in understanding the diversity and commonality of genres within the dataset. Action and role-playing games are the most frequent genres, which indicates their dominance in the indie market.

Missing Values Heatmap A heatmap of missing values was generated to identify and assess the quality of the dataset.

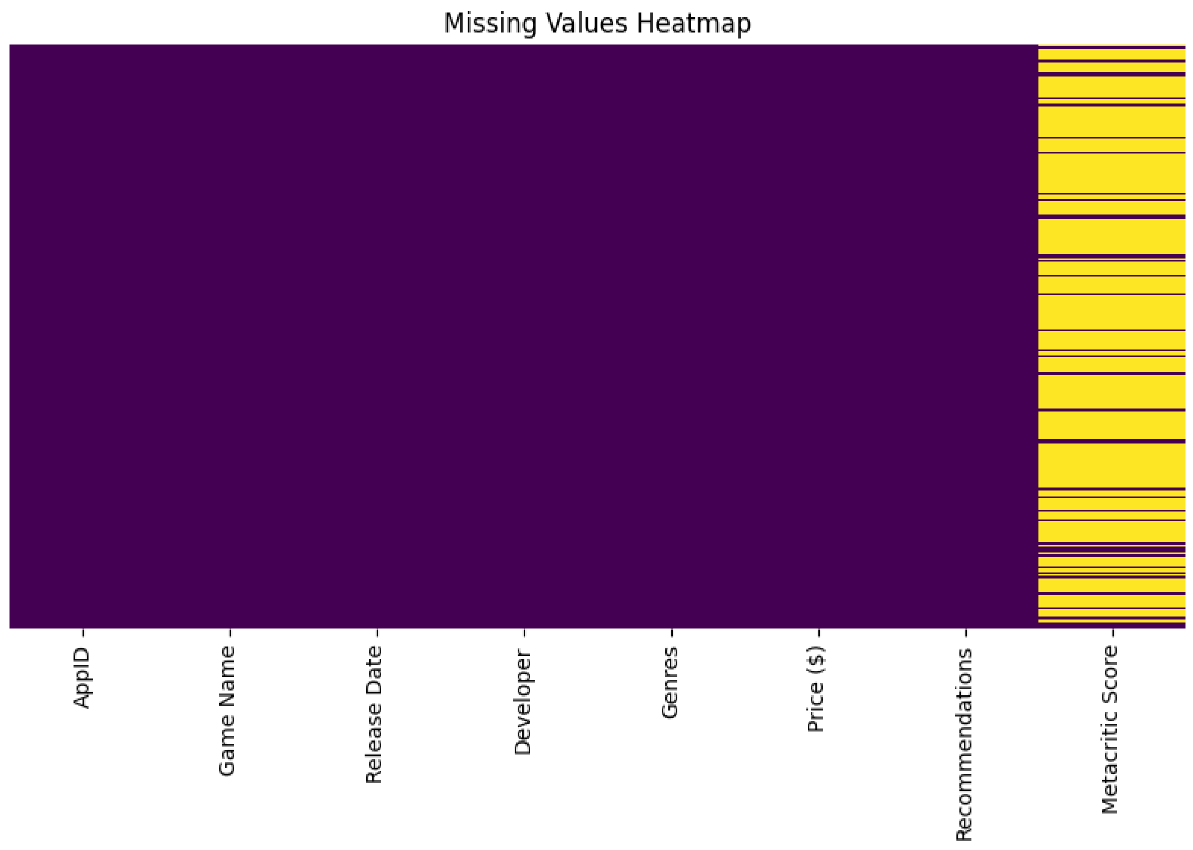


Fig. 12: Missing Values Heatmap. This plot reveals a high concentration of missing values in the 'Metacritic Score' column, while other columns such as 'AppID' and 'Game Name' are mostly complete. Identifying the extent of missing data was crucial for determining appropriate imputation strategies.

Pair Plot of Numerical Features A pair plot was used to visualize relationships between key numerical features, aiding in understanding potential correlations and patterns.

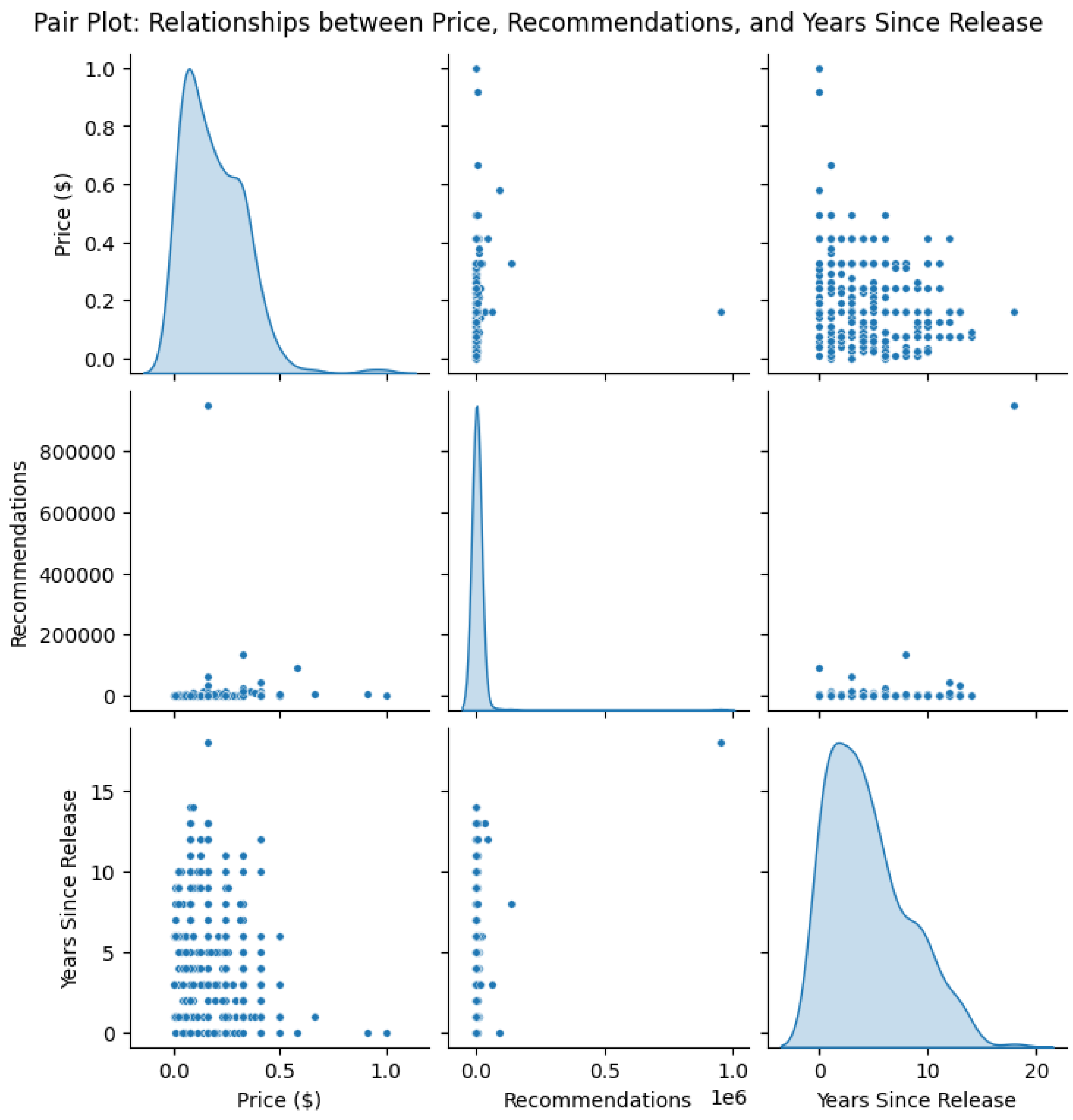


Fig. 13: Pair Plot of Numerical Features. This visualization reveals patterns and correlations between different numerical features in the dataset, such as price, recommendations, and years since release, providing an overview of feature interactions.

Boxplots Boxplots were used to compare the price distributions across different genres, identifying price variations and outliers within each genre.

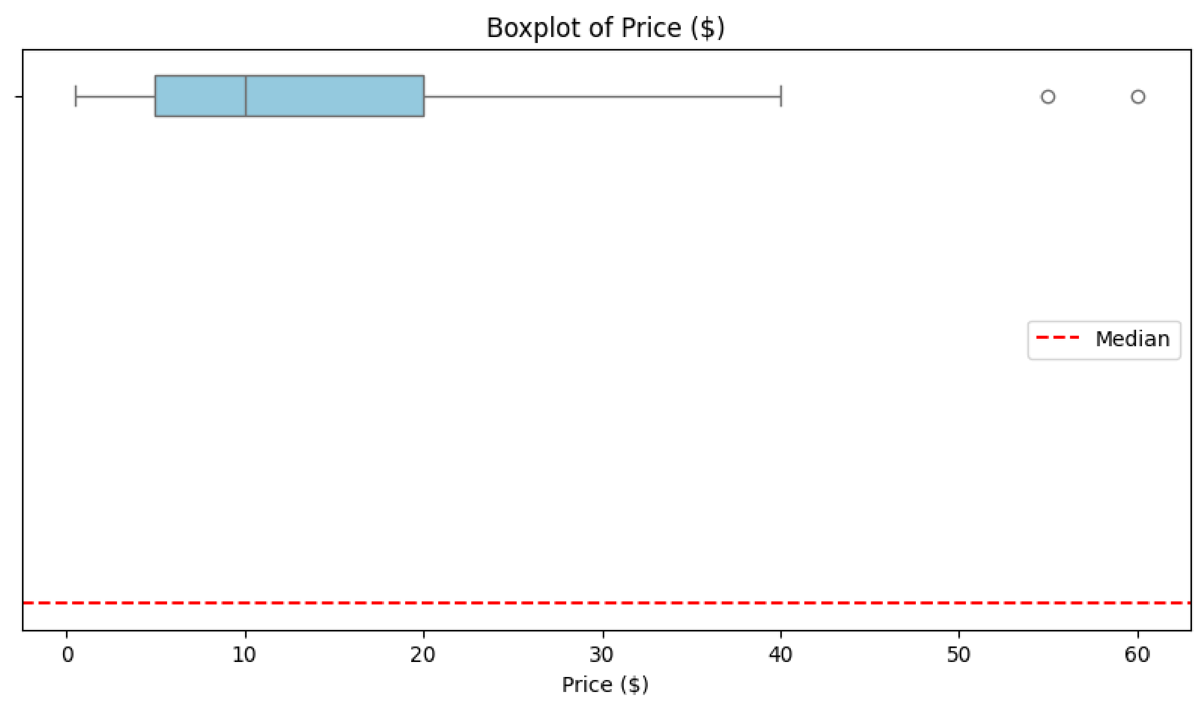


Fig. 14: Boxplot of Game Prices. The plot highlights variations in game pricing and reveals genres with higher average prices, such as role-playing games. This indicates that certain genres may justify higher price points based on perceived value.

Swarmplot of Price Outliers A swarmplot was used to visualize the presence of outliers in the game price data.

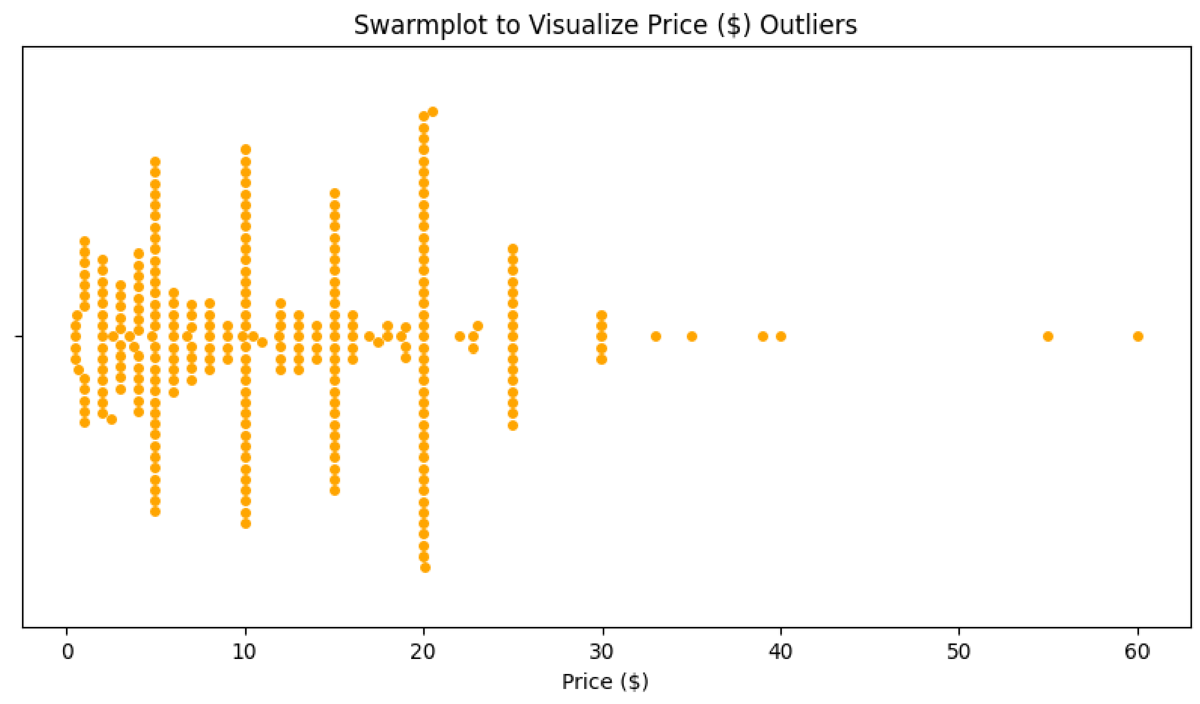


Fig. 15: Swarmplot of Price Outliers. The plot highlights outliers and variability within the game pricing, providing insights into market pricing strategies and identifying unusually priced games.

Years Since Release Price Boxplot A boxplot of years since release against game price was created to understand price variations over time.

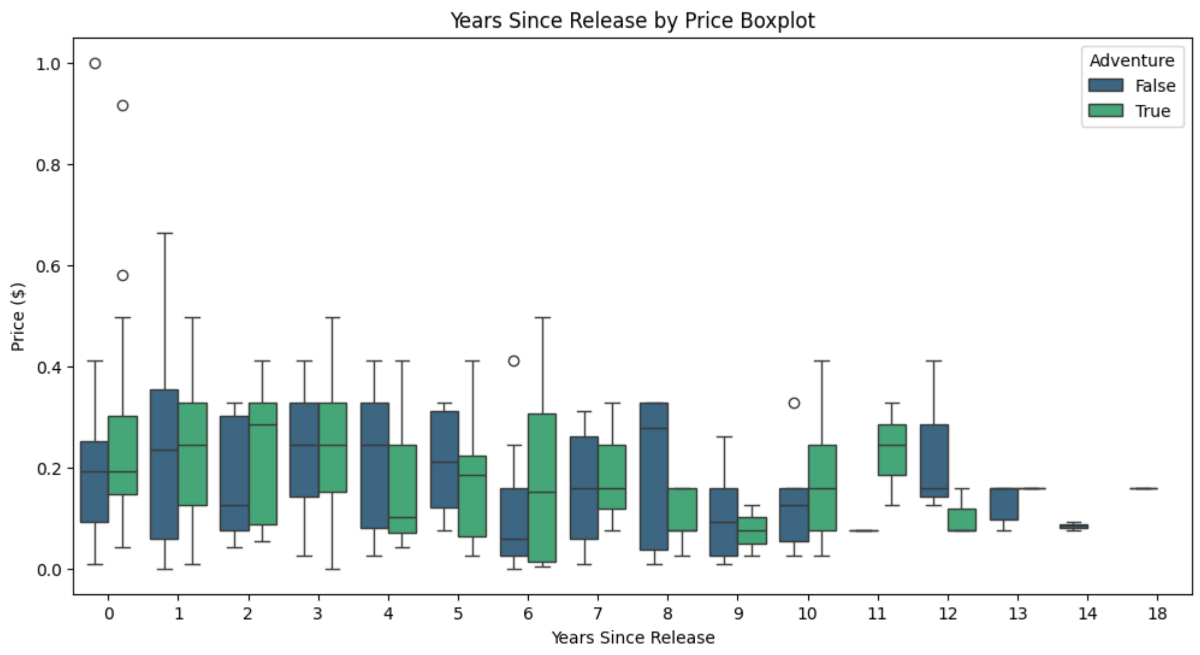


Fig. 16: Boxplot of Years Since Release vs. Price. This visualization helps in understanding how the pricing of games changes as they age. Older games tend to have lower prices, which may be due to discounts or reduced demand over time.

3.3 Findings from EDA

The correlation analysis revealed several notable relationships:

- A moderate positive correlation between the number of recommendations and the price of the game, suggesting that higher-priced games tend to receive more recommendations.
- A strong positive correlation between years since release and recommendations, indicating older games are more likely to accumulate a higher number of recommendations.
- Weak correlations between genres and other features, suggesting that genre alone may not be a strong predictor of game success.
- The presence of outliers in pricing, as indicated by swarmplots, suggests that while most games are clustered around certain price points, a few games are priced significantly higher or lower.

These insights informed the selection of features for model training, focusing on attributes such as recommendations, price, and years since release.

4 Model Development and Performance Assessment

Two machine learning models were developed: Random Forest and Logistic Regression. These models were chosen for their distinct advantages—Random Forest for its robustness and ability to capture non-linear relationships, and Logistic Regression for interpretability.

4.1 Model Development

The dataset was split into training and testing subsets using an 80:20 ratio. Hyperparameter tuning was performed using grid search.

Hyperparameter Tuning Below is the code for hyperparameter optimization for the Random Forest model:

Listing 1.1: Grid Search for Hyperparameter Tuning

```
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import RandomForestClassifier

param_grid = {
    'n_estimators': [100, 200, 300],
    'max_depth': [10, 20, None],
    'min_samples_split': [2, 5, 10]
}

grid_search = GridSearchCV(RandomForestClassifier(), param_grid, cv=3)
grid_search.fit(X_train, y_train)
print("Best␣parameters:", grid_search.best_params_)
```

4.2 Model Evaluation and Feature Importance

As we move into the model evaluation phase, understanding feature importance will play a crucial role in interpreting how each attribute affects the prediction outcome.

Feature Importance The Random Forest model will generate feature importance values that can be visualized using bar charts to show which metadata attributes have the most influence on the model's predictions. This analysis will provide critical insights into attributes like price, number of recommendations, or developer reputation, which could be most influential in determining game success.

5 Conclusions and Future Work

The study aimed to utilize machine learning to predict indie game success based on available metadata. Although model results are pending, significant progress has been made in building a clean dataset, developing meaningful visualizations, and setting up a robust modeling pipeline.

Future Work will focus on executing the models, interpreting the results, and potentially extending the dataset to include additional platforms like Epic Games Store to broaden the analysis scope and improve generalizability.

5.1 Limitations

One limitation noted during the project was the inherent bias in the dataset—games with more exposure or longer availability on the platform naturally accumulate more recommendations, which may not necessarily indicate quality. Additionally, incomplete metadata for some games posed challenges during data cleaning, resulting in some games being excluded from analysis.

Additional Resources

For more details, please refer to the project resources below:

– Overleaf Report
– GitHub Repository
– GitHub Data Directory
– Steam API Documentation

References

1. Kirasich, K., Smith, T., Sadler, B.: Random forest vs logistic regression: Binary classification for heterogeneous datasets. SMU Data Science Review **1**(3), Article 9 (2018), bluehttps://scholar.smu.edu/datasciencereview/vol1/iss3/9, creative Commons License
2. Lounela, K.: On identifying relevant features for a successful indie video game release on steam. Master's Programme in Department of Information and Service Management (2024), bluehttps://aaltodoc.aalto.fi/items/d578980e-71fa-4618-b500-dff30bbac490