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 leveraging game metadata and applying machine learning techniques. Data collected from the Steam API was analyzed using models such as Random Forest, Logistic Regression, and Support Vector Machines (SVM). Key features driving game success, including pricing strategies, release timelines, and gameplay attributes, were identified through feature importance analysis. Exploratory data analysis, including distribution plots and correlation matrices, revealed meaningful relationships between game features and success metrics. The tuned Random Forest model emerged as the most effective predictor, achieving an accuracy of 65.5%, highlighting the significant influence of numerical features like price and years since release. These findings offer actionable insights for indie developers aiming to enhance player engagement and optimize their games for market success.

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 Logistic Regression, Random Forest, and Support Vector Machine (SVM). These models were chosen for their ability to handle complex datasets and provide interpretable insights into the attributes most associated with game popularity. To ensure fairness and avoid bias, the dataset was carefully balanced to represent both popular and lesser-known indie games, providing a more comprehensive understanding of game success. Furthermore, advanced feature engineering techniques were employed, such as transforming skewed features and engineering new features like "Years Since Release" to enrich the model inputs.

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. Challenges in data collection, including Steam API rate limitations, were managed using retry logic and exponential backoff, ensuring a robust dataset for analysis. Model performance was further optimized through hyperparameter tuning using GridSearchCV, allowing for enhanced accuracy and reliability.

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 Logistic Regression, Random Forest, and SVM, 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.

The detailed implementation of the API setup is available in the codebase, which can be accessed here.

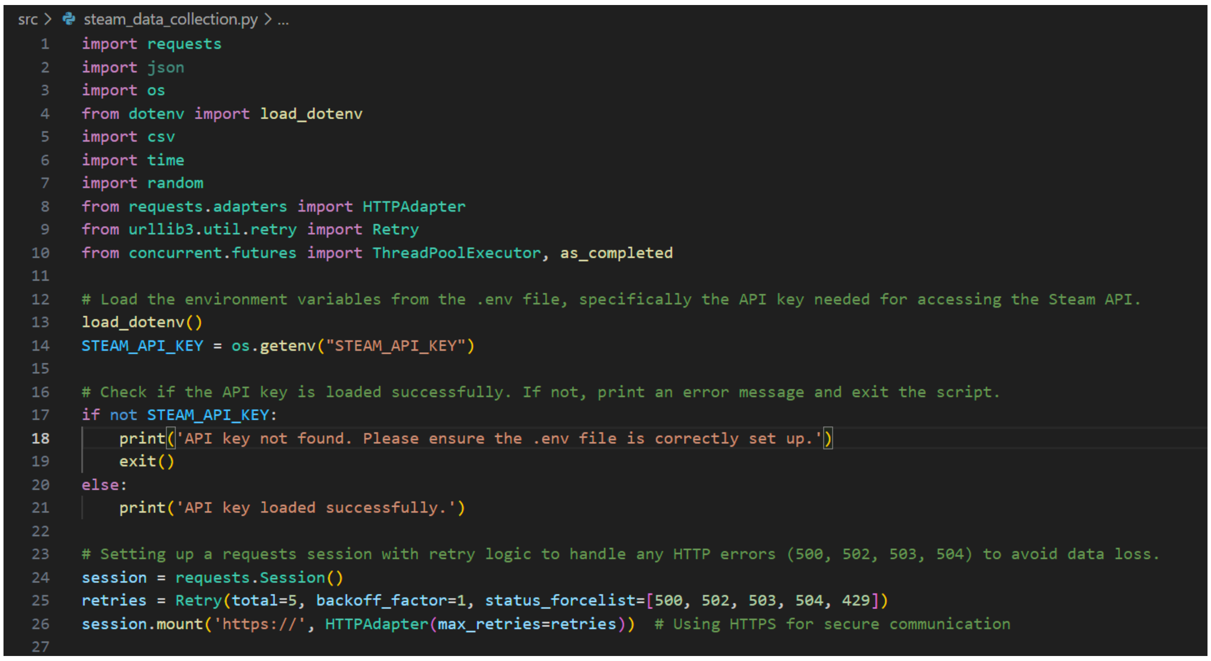
2.1 Source of Data

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. The complete data collection procedure can be found in the [GitHub repository](https://github.com/dgraves4/steam-indie-success).

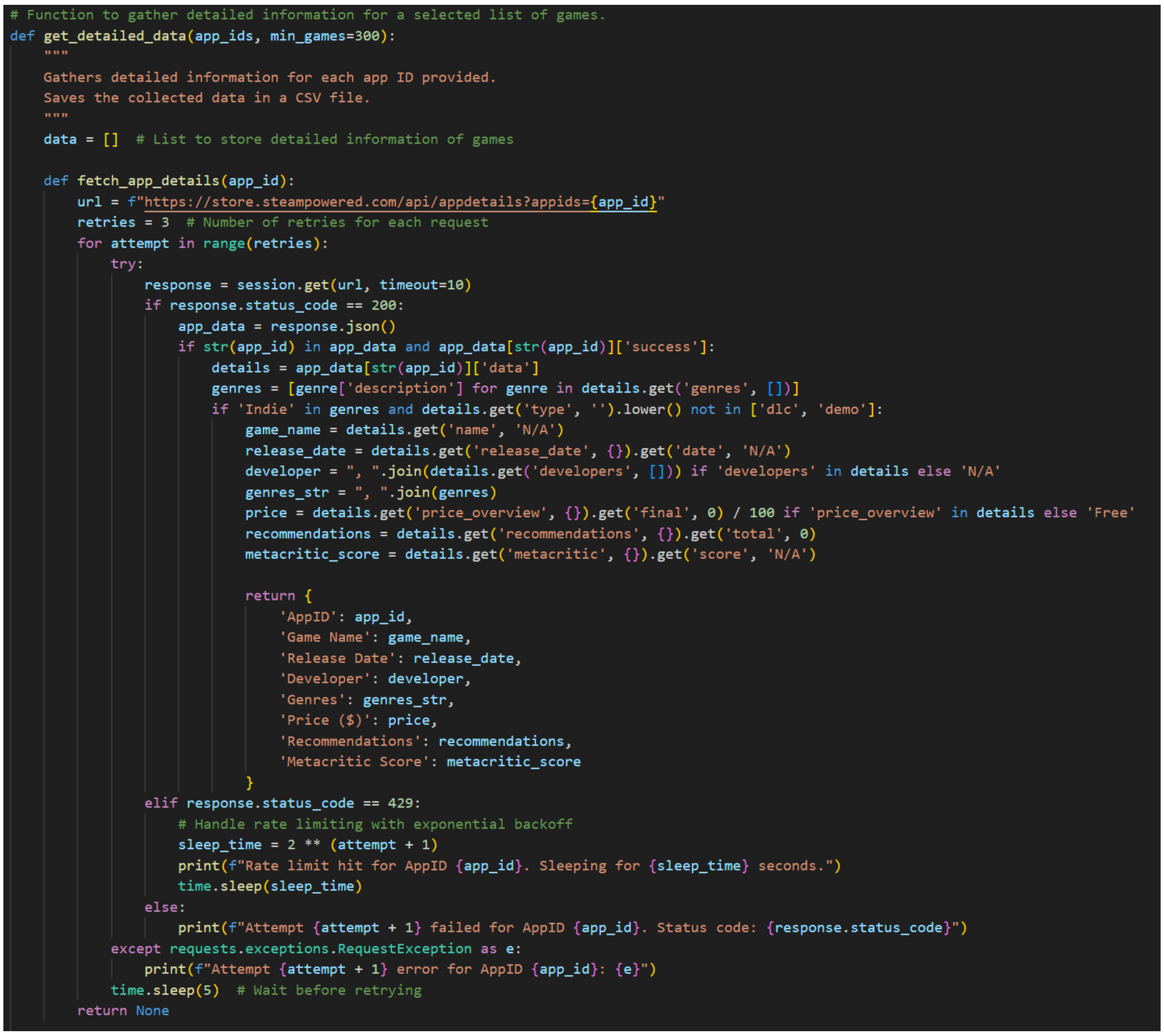
API Setup Environment The Steam Web API was used to collect the required game metadata. Figure 1 shows the Python script used to set up the API environment, providing access to data such as game names, release dates, and game statistics.


src > steam_data_collection.py > ...

```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.

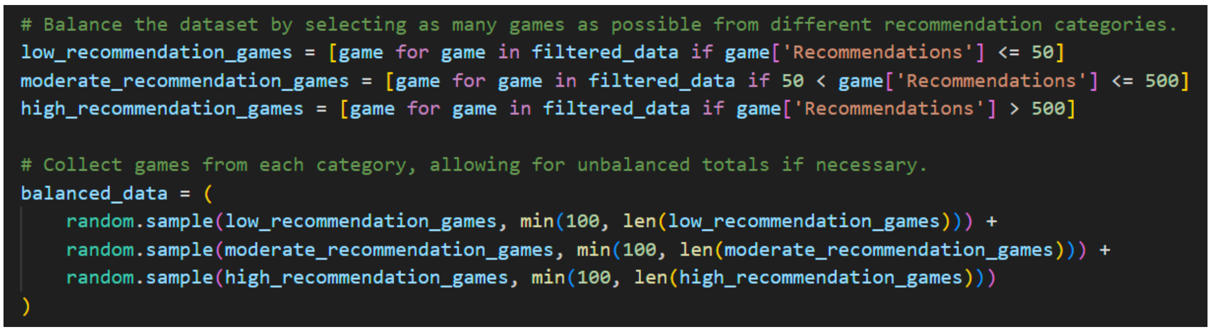
Fetching Detailed Game Data To obtain detailed metadata, multiple API calls were made using the Requests library in Python, with provisions for handling rate limits and retries to manage transient errors. Figure 2 illustrates how the game data was fetched, including a retry mechanism to manage server 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. A retry mechanism using the `Retry` feature from the `urllib3` library ensured resilience against transient errors, such as server timeouts.
- **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.
- **Balanced Dataset:** The dataset was balanced to reduce popularity bias, ensuring a more comprehensive analysis, as shown in Figure 3. Games were selected across varying levels of popularity based on user recommendations.
- **Data Filtering:** To maintain relevance, games meeting specific criteria, such as adult content or early access status, were excluded. This ensured that the dataset focused solely on completed indie games.
- **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 to help mitigate this issue.

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 GitHub Repository

For detailed code implementations, including scripts for API setup, retry logic, data balancing, and further analysis, refer to the steam-indie-success GitHub repository.

3 Data Cleaning and Exploratory Data Analysis (EDA)

The data cleaning and EDA phases aimed to prepare the dataset for predictive modeling by ensuring its quality and exploring its underlying structure. These phases involved systematically filtering out irrelevant or incomplete records, handling missing values appropriately, engineering new features, visualizing the distribution of key attributes, and investigating relationships between features to inform future modeling.

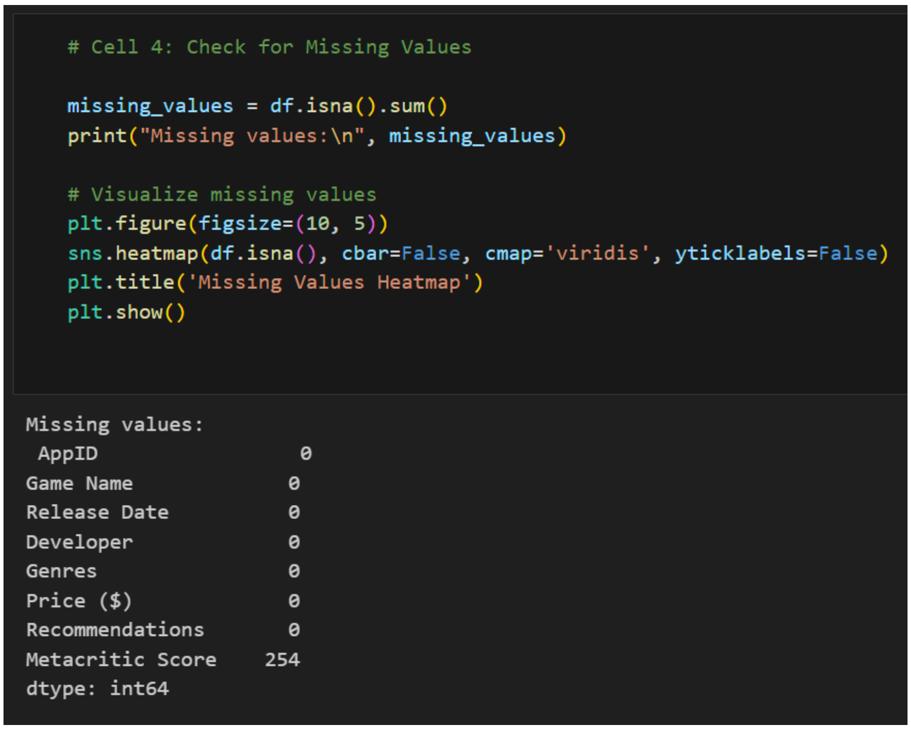
3.1 Data Cleaning Procedure

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:** Figure 4 illustrates a heatmap used to identify columns with missing values. Visualizing missing values helped in determining an effective handling strategy and ensuring that significant data gaps were addressed.

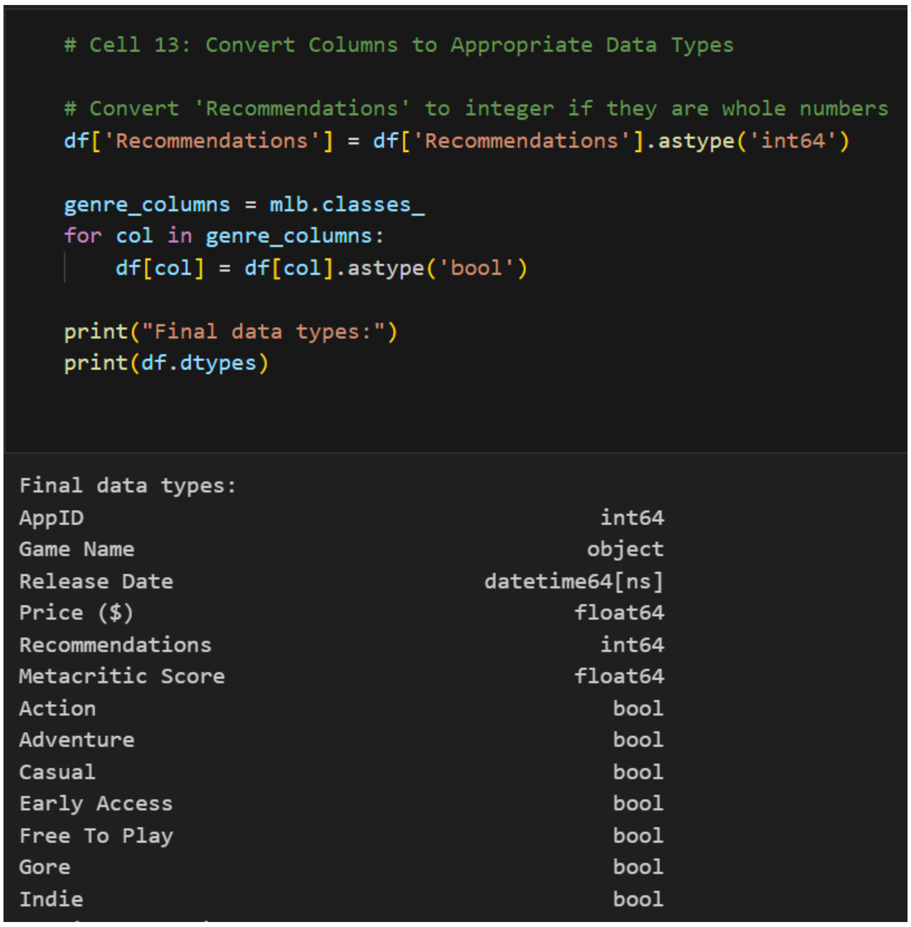
- **Handling Missing Values:** For critical fields such as `Game Name` and `Release Date`, rows with missing values were excluded to maintain data integrity. Missing values in the `Metacritic Score` field were imputed at first, using the median of the column to reduce bias. Later, the Metacritic score feature was fully removed due to a majority of the data missing values. For the `Price ($)` column, data was first converted from strings to numerical values, after which missing entries were also handled appropriately, as shown in Figure 5.
- **Converting Data Types:** To prepare for effective analysis, appropriate data types were assigned to each feature, as shown in Figure 5. For example, `Release Date` was converted to a datetime type for proper temporal analysis, numerical features stored as strings (e.g., `Price ($)`) were converted to floats, and categorical fields such as game genres and developers were transformed using one-hot encoding to facilitate analysis and model training.
- **Feature Engineering:** Figure 6 depicts the feature engineering process. The feature `Years Since Release` was created to capture the temporal aspect of game popularity by converting `Release Date` into a numerical format and subtracting it from 2024. This feature helps in understanding the impact of a game's age on its success. In addition, categorical variables like game genres were one-hot encoded, as illustrated in Figure 7, to improve their representation in the models without imposing an ordinal relationship.
- **Normalization:** To ensure that features contributed equally to model performance, numerical columns such as `Price ($)` were scaled using Min-Max normalization, as shown in Figure 8. Min-Max normalization was chosen to maintain the interpretability of the features within the range of 0 to 1, ensuring effective gradient calculations during model training.
- **Identifying and Handling Outliers:** Outliers in numerical features, such as `Price ($)`, were identified using the interquartile range (IQR) method. Extreme outliers were capped or removed to mitigate their influence on model training. This approach reduced skewness and ensured the dataset was more representative of common trends.
- **Removing Duplicate Rows:** Duplicate records were removed to ensure data uniqueness, resulting in a cleaner dataset for analysis. This helped prevent redundant information from affecting the modeling phase.
- **Log Transformation of Recommendations:** The `Recommendations` column was log-transformed to address skewness and stabilize variance, making it more suitable for modeling.
- **Standardization of Numerical Features:** Features such as `Recommendations` and `Years Since Release` were standardized to ensure that each feature contributed equally during model training, facilitating better gradient-based optimization.
- **Dropping Columns with High Missing Values:** The `Metacritic Score` column was dropped since approximately 85% of its values were missing, making it unreliable for analysis.



```
# 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
```

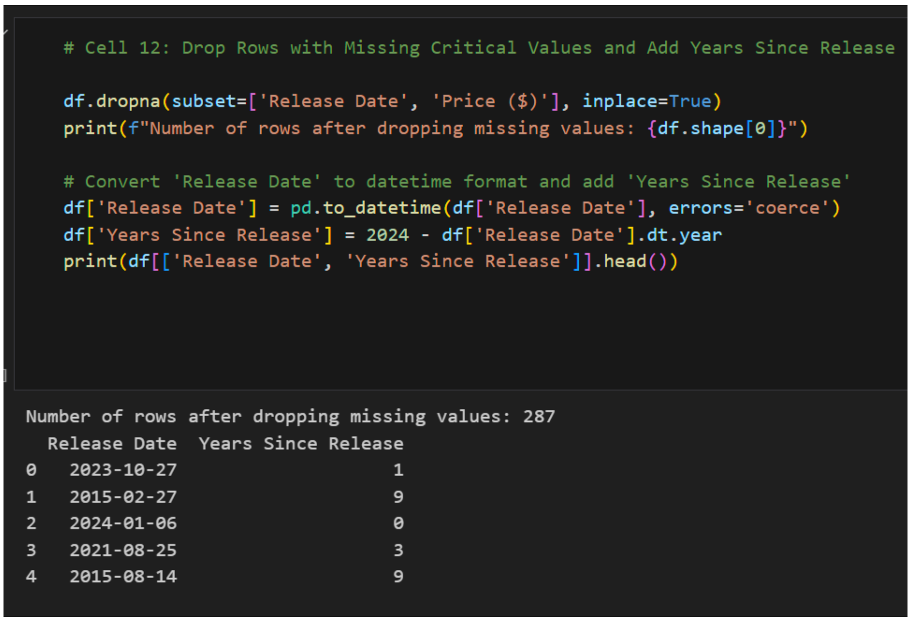
Fig. 4: Identifying missing values using a heatmap to assess the data quality.



```
# 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
```

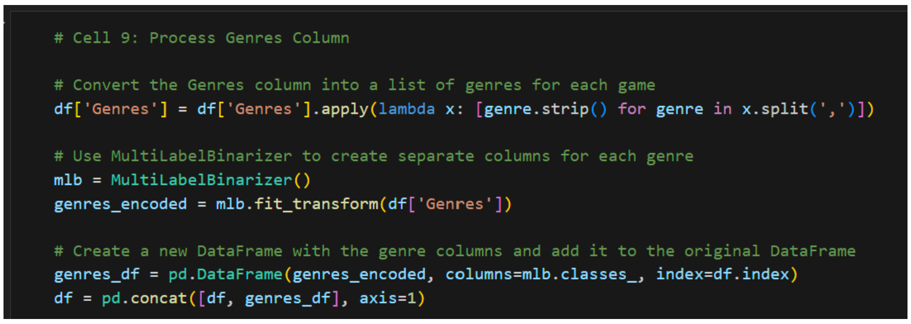
Fig. 5: Handling missing values and converting data types for better usability.



```
# 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())
```

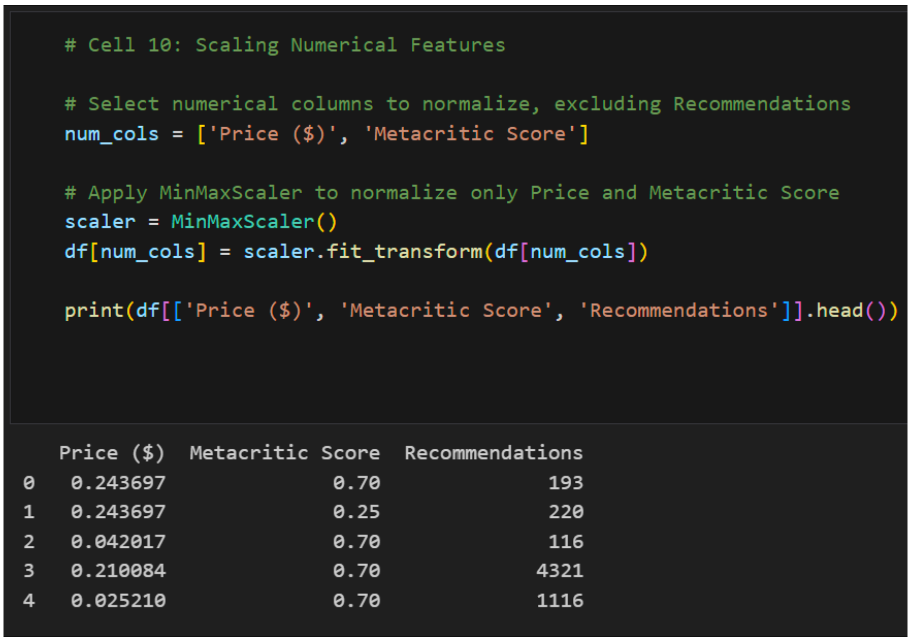
```
Number of rows after dropping missing values: 287
  Release Date  Years Since Release
0   2023-10-27                    1
1   2015-02-27                    9
2   2024-01-06                    0
3   2021-08-25                    3
4   2015-08-14                    9
```

Fig. 6: Feature engineering of `Years Since Release` by transforming the `Release Date` to provide temporal context.



```
# 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 of game genres to facilitate effective categorical representation in models.



```
# 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 to ensure consistent feature scaling.

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.

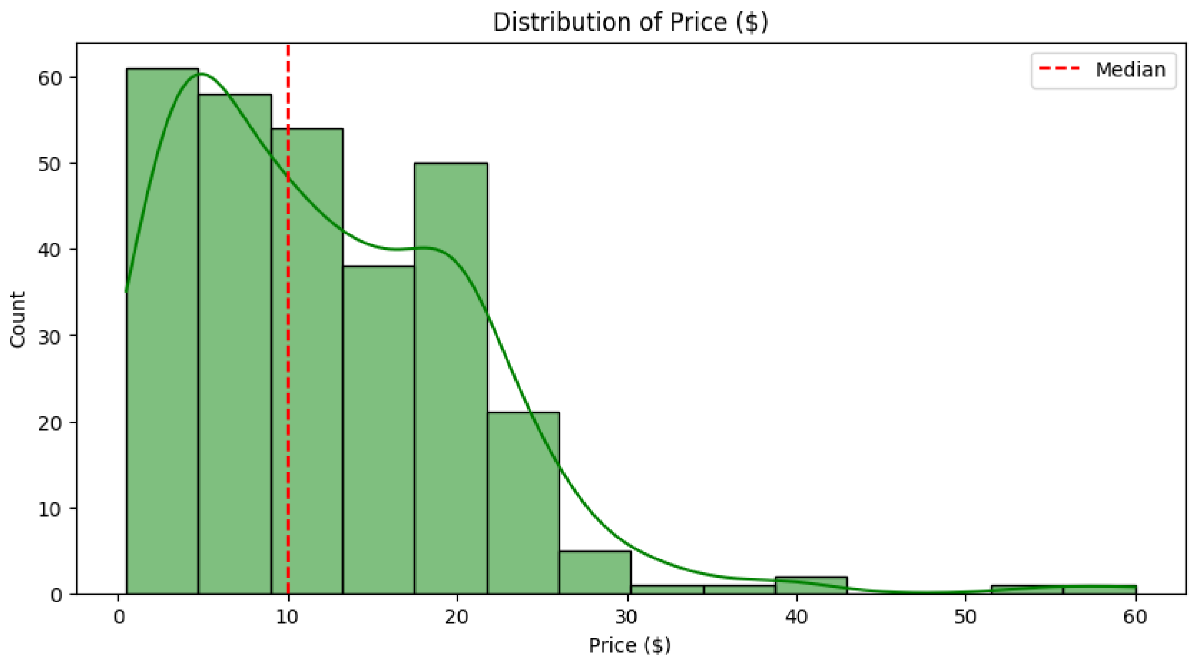
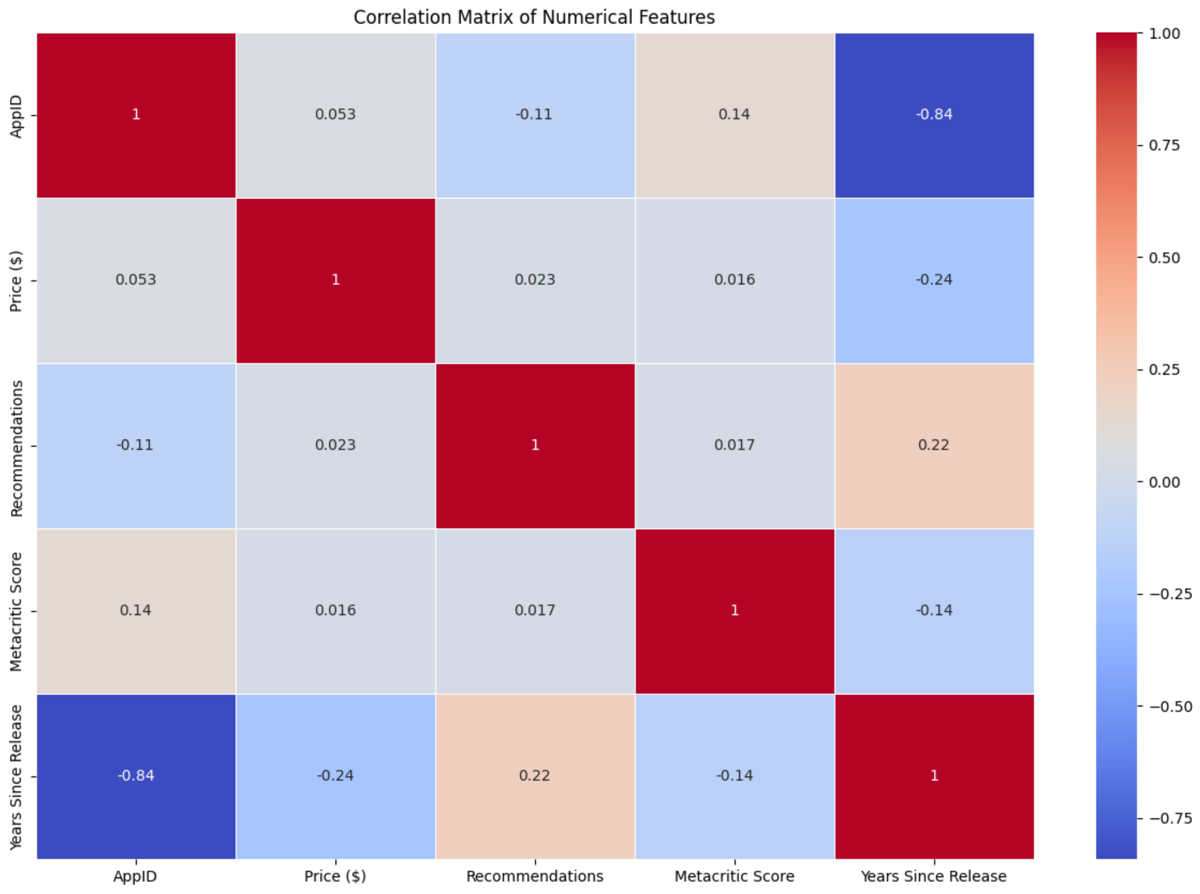


Fig. 9: Distribution of Game Prices. The distribution reveals that most indie games are priced below $20, with some outliers at higher price points. This suggests that indie developers generally adopt a low-price strategy to attract a wider audience. However, the presence of outliers indicates that some developers target markets willing to pay a premium. These pricing strategies could reflect perceived value, production quality, or brand reputation.

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.



	AppID	Price ($)	Recommendations	Metacritic Score	Years Since Release
AppID	1	0.053	-0.11	0.14	-0.84
Price ($)	0.053	1	0.023	0.016	-0.24
Recommendations	-0.11	0.023	1	0.017	0.22
Metacritic Score	0.14	0.016	0.017	1	-0.14
Years Since Release	-0.84	-0.24	0.22	-0.14	1

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', likely due to the way that Steam assigns ID numbers over time. 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, and other factors such as genre or gameplay quality may have a more significant influence. 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.

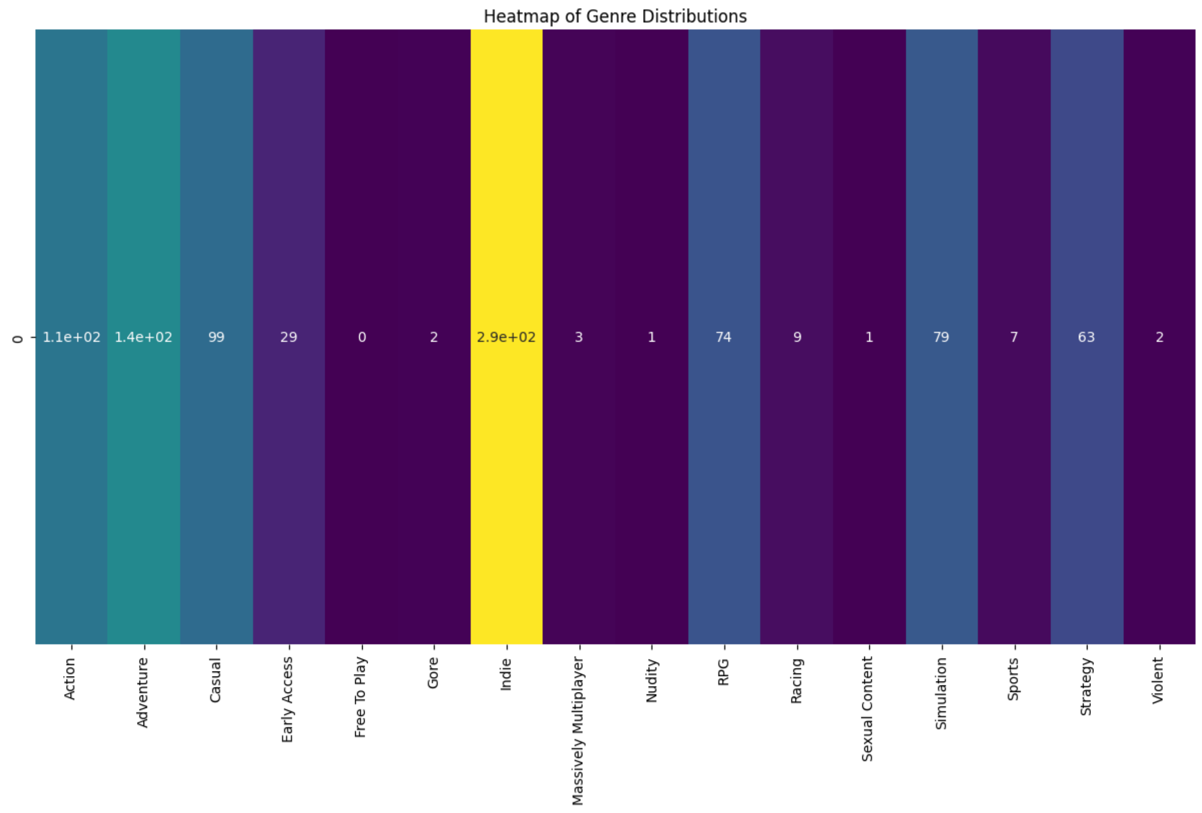


Fig. 11: Heatmap of Genre Distributions. This plot helps in understanding the diversity and commonality of genres within the dataset. Action and role-playing games are the most frequent genres, which indicates their dominance in the indie market. This dominance may be due to the popularity of these genres among players or the lower barriers to developing action and role-playing games compared to genres like simulation or strategy. Understanding this trend can help developers align their game concepts with market demand.

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

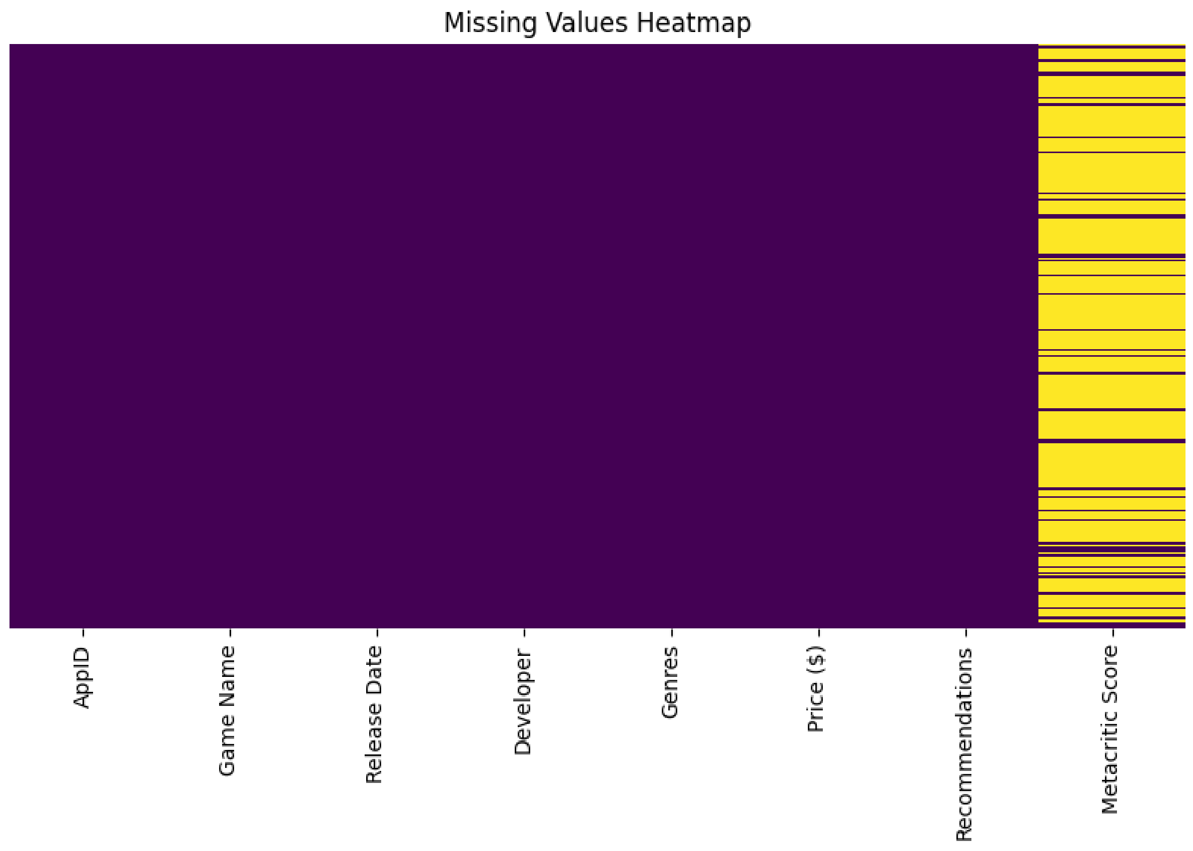


Fig. 12: Missing Values Heatmap. This plot reveals a high concentration of missing values in the 'Metacritic Score' column, while other columns such as 'AppID' and 'Game Name' are mostly complete. Identifying the extent of missing data was crucial for determining appropriate imputation strategies, and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

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

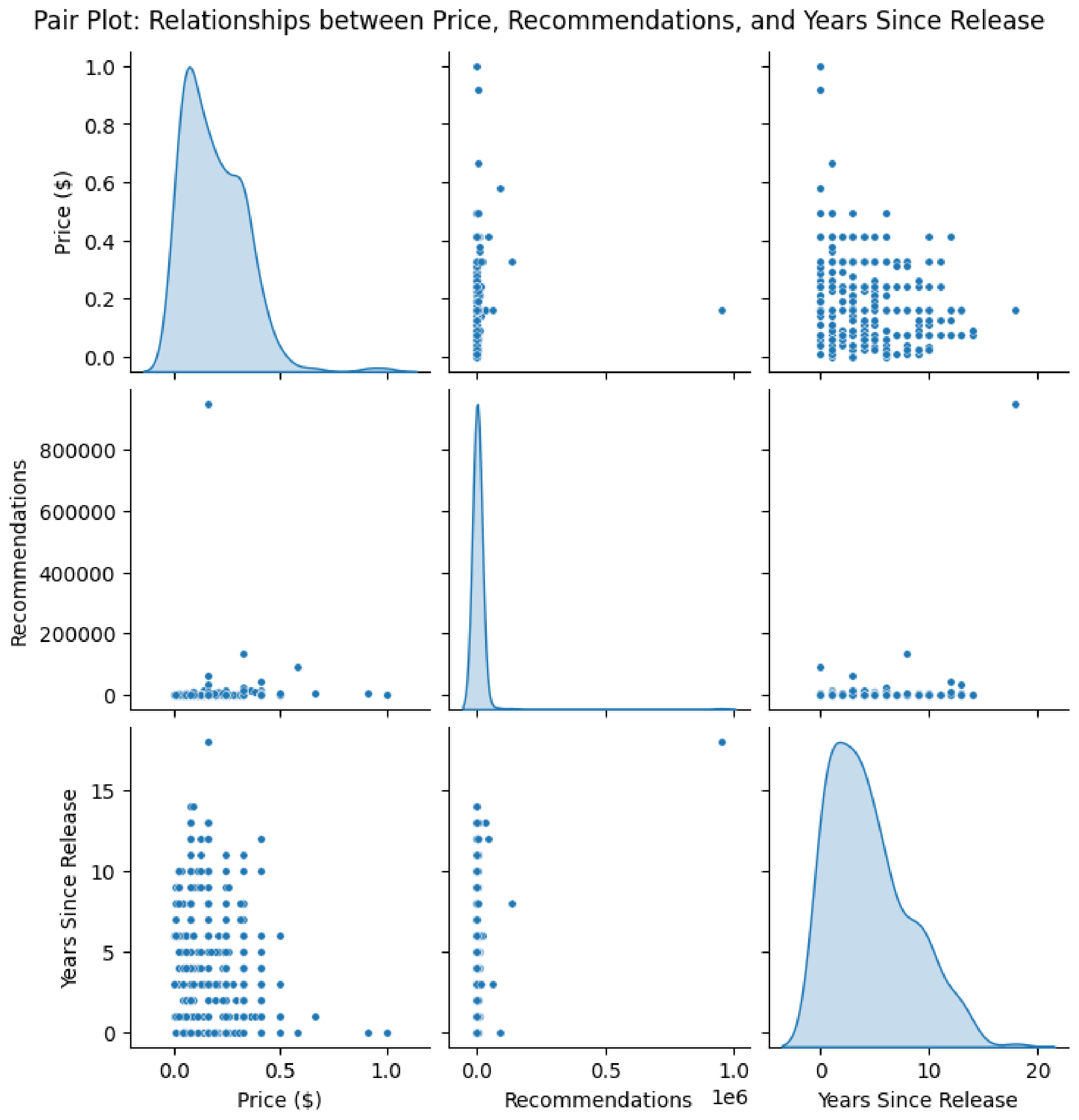


Fig. 13: Pair Plot of Numerical Features. The pair plot provides a comprehensive view of relationships between key numerical features, such as price, recommendations, and years since release. It highlights trends and potential correlations, such as a clustering of low-price games receiving a high number of recommendations, suggesting affordability may play a role in player engagement. Additionally, the spread of recommendations across 'Years Since Release' indicates that older games have had more time to accumulate player feedback. By visualizing these interactions, the plot aids in identifying features that may influence the success of indie games and informs hypothesis generation for modelings.

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

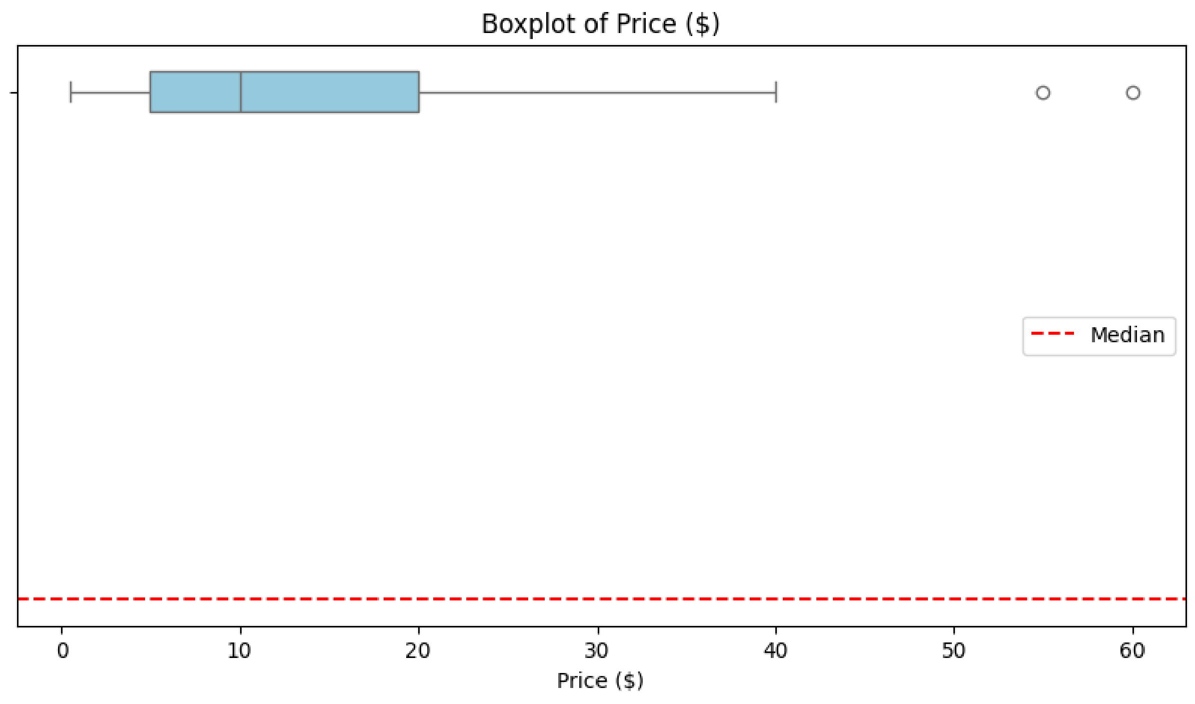


Fig. 14: Boxplot of Game Prices. This visualization displays the distribution of indie game prices on Steam, highlighting that the majority are priced below 20 units. Outliers above 50 units indicate the presence of a few premium-priced games.

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

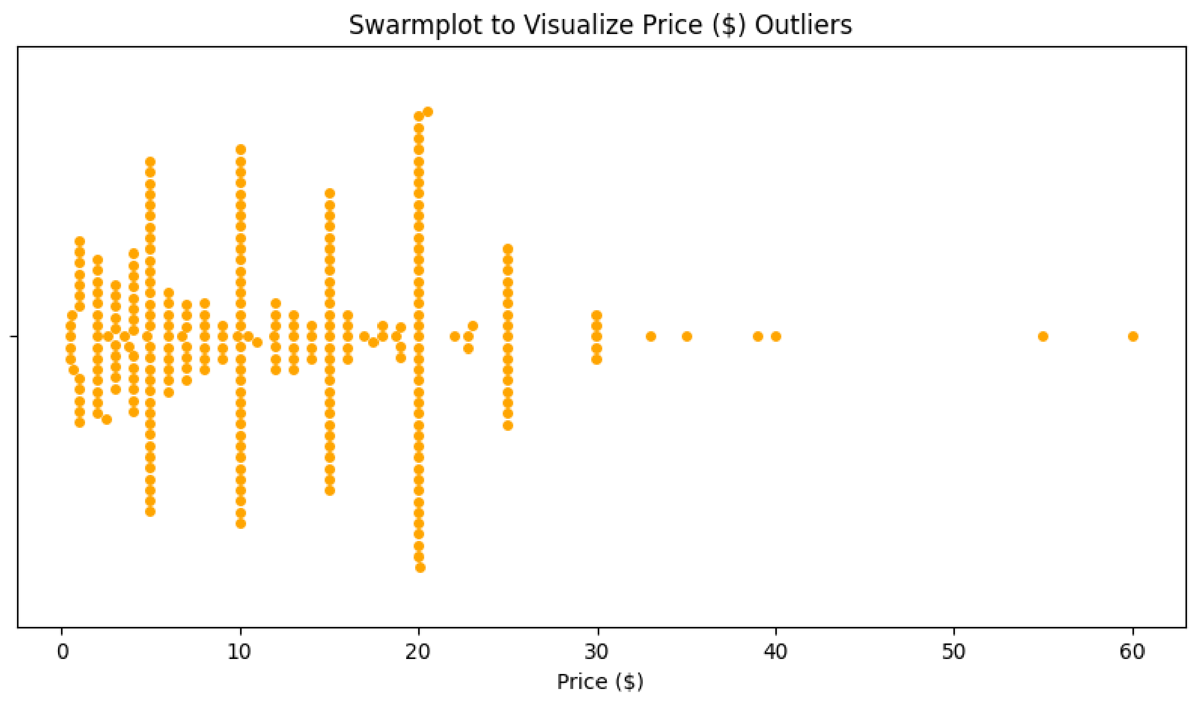


Fig. 15: Swarmplot of Price Outliers. The plot highlights outliers and variability within the game pricing, providing insights into market pricing strategies and identifying unusually priced games.

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

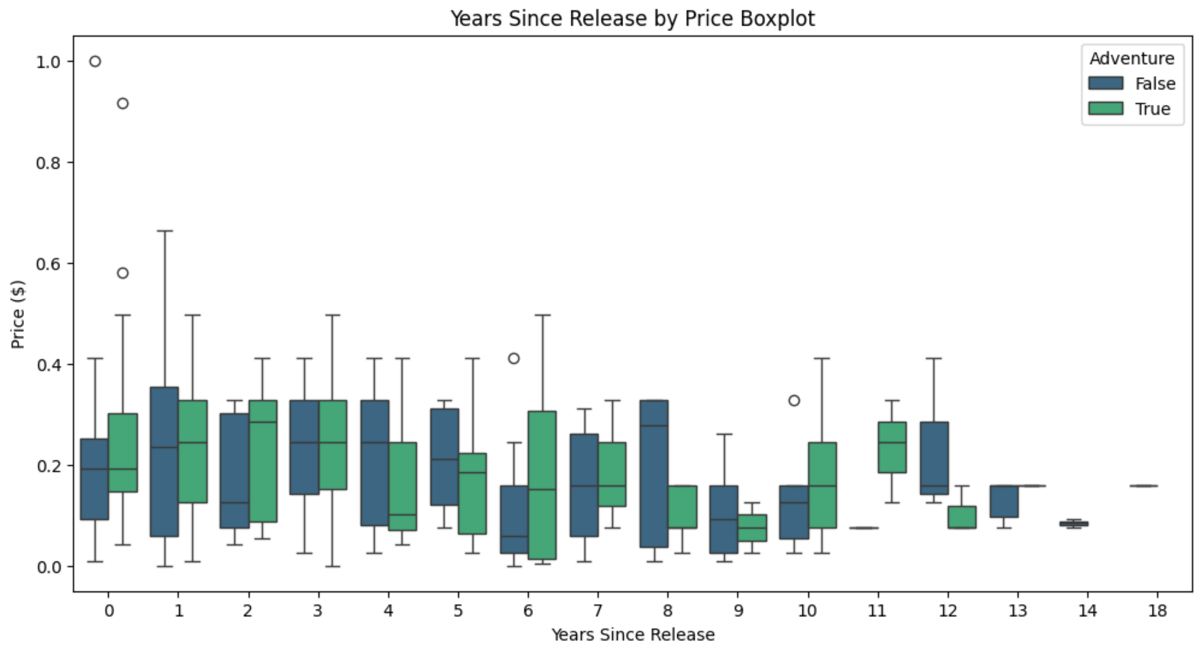


Fig. 16: This plot illustrates the relationship between game prices and the time since release, highlighting that older games generally see lower prices due to discounts or reduced demand. The inclusion of 'Adventure' games reveals slight variations in pricing trends, with these games occasionally retaining higher prices compared to others. Outliers indicate premium titles maintaining higher value over time.

3.3 Findings from EDA

The exploratory data analysis revealed several significant relationships and insights that informed feature selection and guided the subsequent modeling phase. Key findings include:

Correlation Insights

- A strong positive correlation exists between years since release and recommendations. This trend underscores the cumulative nature of recommendations: older games have had more time to garner reviews and player engagement.
- No significant correlation was observed between price and recommendations, suggesting that price alone does not strongly influence player feedback.

Genre and Feature Relationships

- Weak correlations were found between genres and other features, such as price or recommendations. This suggests that genre alone is not a strong predictor of success. Instead, gameplay quality, developer reputation, or other factors likely play a more significant role.

Pricing Trends and Outliers

- Boxplots and swarmplots revealed that most indie games are priced below $20, reflecting an affordability strategy to attract a larger audience. Outliers indicate premium pricing for high-value or niche games, as well as occasional promotional pricing.
- Over time, game prices tend to decline, as shown in the boxplot of years since release versus price. This aligns with common industry practices where older titles are discounted to maintain sales and relevance.
- Pricing outliers further reflect experimental strategies, exclusivity, or highly anticipated titles commanding premium prices.

Feature Selection for Modeling From these findings, recommendations, price, and years since release emerged as key features for modeling game success. These attributes are associated with player engagement and revenue trends, with ”Years Since Release” introducing a critical temporal perspective.

4 Model Development and Performance Assessment

Three machine learning models were developed: Logistic Regression, Random Forest, and Support Vector Machine (SVM). These models were chosen for their complementary strengths—Logistic Regression for simplicity, Random Forest for robustness, and SVM for flexibility with kernels and handling class imbalances.

4.1 Data Splitting

The target variable for this project was a binary classification, where a game was labeled as successful (`1`) if it had more than 500 recommendations and unsuccessful (`0`) otherwise. Figure 17 shows the distribution of the target variable, revealing a class imbalance, with more samples classified as unsuccessful (`0`).

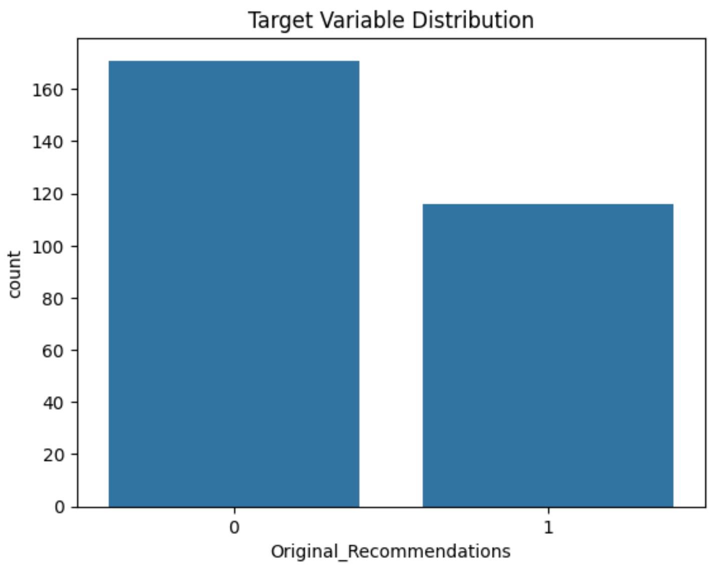


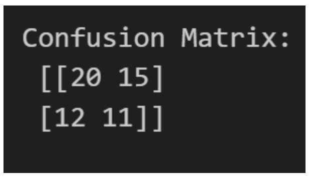
Fig. 17: Distribution of the Binary Target Variable. The chart illustrates the class imbalance, with more samples classified as non-successful (`0`) than successful (`1`). Stratified sampling was used to maintain this distribution in training and testing sets.

4.2 Baseline Model Performance

Logistic Regression

Using ‘Years Since Release‘ only:

- **Accuracy**: 53.4%
- **Precision**: Class 0: 62%, Class 1: 42%
- **Recall**: Class 0: 57%, Class 1: 48%
- **F1-Score**: 52%

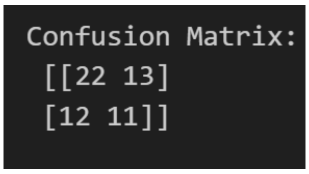


```
Confusion Matrix:
 [[20 15]
 [12 11]]
```

Fig. 18: Confusion Matrix for Logistic Regression (Years Since Release only).

Using Combined Features:

- **Accuracy**: 56.9%
- **Precision**: Class 0: 65%, Class 1: 46%
- **Recall**: Class 0: 63%, Class 1: 48%
- **F1-Score**: 55%



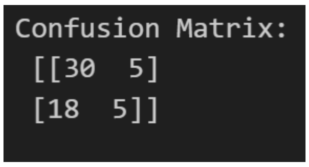
```
Confusion Matrix:
 [[22 13]
 [12 11]]
```

Fig. 19: Confusion Matrix for Logistic Regression (Combined Features).

Random Forest (Untuned and Tuned)

Using ‘Years Since Release‘ only (Untuned):

- **Accuracy**: 60.3%
- **Precision**: Class 0: 62%, Class 1: 50%
- **Recall**: Class 0: 86%, Class 1: 22%
- **F1-Score**: 51%

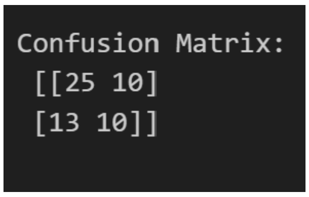


```
Confusion Matrix:
 [[30  5]
 [18  5]]
```

Fig. 20: Confusion Matrix for Random Forest (Years Since Release only).

Using Combined Features (Untuned):

- **Accuracy**: 60.3%
- **Precision**: Class 0: 66%, Class 1: 50%
- **Recall**: Class 0: 71%, Class 1: 43%
- **F1-Score**: 58%



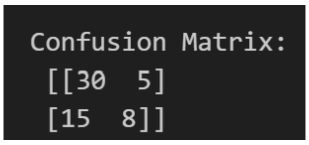
```
Confusion Matrix:
 [[25 10]
 [13 10]]
```

Fig. 21: Confusion Matrix for Random Forest (Untuned, Combined Features).

Using Combined Features (Tuned):

- **Best Parameters**: `max_depth=10`, `n_estimators=100`, `min_samples_split=10`, `min_samples_leaf=4`, `class_weight=None`.
- **Accuracy**: 65.5%

- **Precision**: Class 0: 67%, Class 1: 62%
- **Recall**: Class 0: 86%, Class 1: 35%
- **F1-Score**: 60%



```
Confusion Matrix:
 [[30  5]
 [15  8]]
```

Fig. 22: Confusion Matrix for Tuned Random Forest (Combined Features).

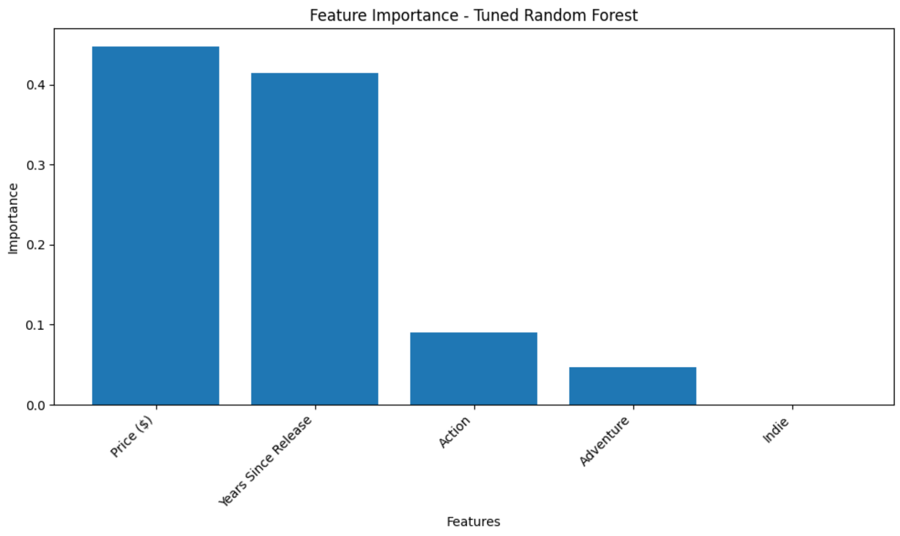
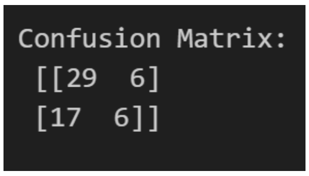


Fig. 23: Feature Importance for Tuned Random Forest (Combined Features). This visualization highlights that `Price` and `Years Since Release` were the most influential features in predicting game recommendations.

Support Vector Machine (Untuned and Tuned)

Using ‘Years Since Release‘ only (Untuned):

- **Accuracy**: 60.3%
- **Precision**: Class 0: 63%, Class 1: 50%
- **Recall**: Class 0: 83%, Class 1: 26%
- **F1-Score**: 53%

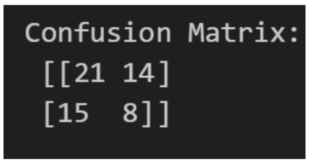


```
Confusion Matrix:
 [[29  6]
 [17  6]]
```

Fig. 24: Confusion Matrix for SVM (Years Since Release only).

Using Combined Features (Untuned):

– **Accuracy**: 50.0%
– **Precision**: Class 0: 58%, Class 1: 36%
– **Recall**: Class 0: 60%, Class 1: 35%
– **F1-Score**: 47%

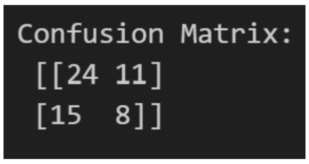


```
Confusion Matrix:
 [[21 14]
 [15  8]]
```

Fig. 25: Confusion Matrix for Untuned SVM (Combined Features).

Using Combined Features (Tuned):

– **Best Parameters**: `kernel='poly'`, `C=1`, `gamma='auto'`.
– **Accuracy**: 55.2%
– **Precision**: Class 0: 62%, Class 1: 42%
– **Recall**: Class 0: 69%, Class 1: 35%
– **F1-Score**: 51%



```
Confusion Matrix:
 [[24 11]
 [15  8]]
```

Fig. 26: Confusion Matrix for Tuned SVM (Combined Features).

4.3 Hyperparameter Tuning Process

To improve the performance of the Random Forest and SVM models, hyperparameter tuning was conducted using `GridSearchCV`. This process systematically searches through a predefined grid of hyperparameters to find the combination that yields the best performance. For the Random Forest model, parameters such as the number of estimators (`n_estimators`), maximum tree depth (`max_depth`), minimum samples per leaf (`min_samples_leaf`), and class weights (`class_weight`) were optimized. Similarly, for the SVM model, parameters like the kernel type (`kernel`), penalty parameter (`C`), and gamma (`gamma`) were explored.

The following Python code demonstrates the setup of the parameter grid and the execution of `GridSearchCV` for hyperparameter tuning of the SVM model. A similar approach was applied to the Random Forest model.

Listing 1.1: SVM Hyperparameter Tuning Code

```
param_grid_svm = {
    'C': [0.1, 1, 10, 100],
    'kernel': ['linear', 'rbf', 'poly', 'sigmoid'],
    'gamma': ['scale', 'auto']
}
svm_grid_search = GridSearchCV(SVC(random_state=42, class_weight='balanced')
    ,
                               param_grid_svm, cv=5, scoring='accuracy')
svm_grid_search.fit(X_train, y_train)
print("Best␣Parameters␣for␣SVM:", svm_grid_search.best_params_)
```

The best hyperparameters found for each model are as follows:

- **Random Forest:** `max_depth=10`, `n_estimators=100`, `min_samples_split=10`, `min_samples_leaf=4`, `class_weight=None`.
- **SVM:** `kernel='poly'`, `C=1`, `gamma='auto'`.

The results of hyperparameter tuning led to significant improvements in the performance of the Random Forest model, as detailed in Section 23. In contrast, the SVM model's performance saw only marginal gains, highlighting the importance of selecting suitable algorithms for this dataset.

4.4 Model Comparison and Key Findings

Table 1 summarizes the performance of the models on the test dataset, including accuracy, precision, recall, and F1-score for the majority class (Class 0) and minority class (Class 1).

Table 1: Model Performance Summary

Model	**Features**	**Accuracy**	**Precision (C0)**	**Recall (C0)**	**Precision (C1)**	**Recall (C1)**	**F1-Score**
Logistic Regression	Years Since Release	53.4%	62%	57%	42%	48%	52%
Logistic Regression	Combined Features	56.9%	65%	63%	46%	48%	55%
Random Forest (Untuned)	Years Since Release	60.3%	62%	86%	50%	22%	51%
Random Forest (Tuned)	Combined Features	65.5%	67%	86%	62%	35%	60%
SVM (Untuned)	Years Since Release	60.3%	63%	83%	50%	26%	53%
SVM (Tuned)	Combined Features	55.2%	62%	69%	42%	35%	51%

The evaluation results are summarized below:

- **Best Model:** The Tuned Random Forest model outperformed all other models with an accuracy of 65.5% and higher precision and recall for both classes compared to the other models. This indicates that the model was effective in capturing the patterns in the data.
- **Insights from Feature Importance:** The Random Forest feature importance chart (Figure 23) revealed that numerical features like `Price` and `Years Since Release` were the most significant predictors. These features contributed more to the predictions than categorical features like `Action` or `Indie`.
- **Logistic Regression Performance:** While simpler, Logistic Regression achieved moderate performance with combined features, achieving an accuracy of 56.9%. However, its limited flexibility in capturing nonlinear relationships affected its overall predictive power.
- **SVM Limitations:** The tuned SVM model showed only marginal improvement over its untuned version. Its accuracy of 55.2% and relatively low recall for Class 1 indicate that SVM may not be the most suitable algorithm for this dataset, particularly for handling class imbalances.
- **Class Imbalance Challenges:** Despite the balanced class weights applied during hyperparameter tuning, the recall for Class 1 (minority class) remained significantly lower across all models. This suggests that further techniques, such as oversampling or synthetic data generation (e.g., SMOTE), may be required to improve minority class prediction.

These results highlight the importance of hyperparameter tuning and feature selection in improving model performance. The findings suggest that while the Random Forest model achieved the best overall performance, additional efforts to address class imbalance and explore advanced feature engineering could further enhance predictive accuracy.

The full implementation of the predictive analysis, including the code, results, and notebooks, can be found in the project repository: Predictive Analysis Notebook.

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 and Challenges

While the models provided valuable insights, several limitations were noted:

- **Class Imbalance**: The dataset showed a higher representation of Class 0 compared to Class 1, affecting model performance, particularly recall for Class 1.
- **Feature Simplification**: Using 'Years Since Release' alone was insufficient for capturing complex relationships in the data, as evidenced by the lower performance of Logistic Regression and SVM with this feature set.
- **Computational Complexity**: Hyperparameter tuning for Random Forest and SVM required significant computational resources, especially for the larger parameter grids.

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), https://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), https://aaltodoc.aalto.fi/items/d578980e-71fa-4618-b500-dff30bbac490