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. The indie game market on Steam is thriving, with thousands of titles competing for attention each year. Yet, many developers struggle to understand what truly drives success. This study combines game metadata with machine learning techniques to predict the success of indie games on Steam and provide actionable insights for developers. Leveraging data from the Steam Web API, we built models like Random Forest, Logistic Regression, and Support Vector Machines to analyze key factors, including pricing strategies, release timelines, and gameplay attributes. Our tuned Random Forest model achieved 65.5% accuracy, revealing that features like pricing and years since release play a pivotal role in success. By addressing challenges such as dataset imbalance and missing values, this research not only improves predictive accuracy but also offers practical strategies for developers to enhance player engagement and optimize their games for market success. Whether you're an indie developer or a curious gamer, our findings shed light on what it takes to succeed in the competitive gaming marketplace.

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

Steam, the world's largest PC gaming platform, hosts tens of thousands of games, with new titles added weekly. For indie developers, this presents both a tremendous opportunity and a major challenge. In a marketplace saturated with releases, the question is no longer just about creating a great game—it's about understanding the factors that drive success. What makes one game rise to the top of the charts while others fade into obscurity? For developers with limited budgets and resources, cracking this code is crucial.

This study seeks to illuminate what defines success for indie games on Steam by analyzing metadata with machine learning techniques. Using the Steam Web API, we compiled a dataset of indie games spanning more than a decade. Models such as Logistic Regression, Random Forest, and Support Vector Machine (SVM) were applied to uncover key predictors like price, recommendations, and years since release. To reduce bias, the dataset was balanced to represent both highly popular and lesser-known games, ensuring a broad perspective.

Exploratory data analysis (EDA) helped uncover patterns using techniques like distribution plots and correlation matrices. Advanced feature engineering introduced variables such as "Years Since Release" to capture time-based performance factors. Challenges such as Steam API rate limits were addressed with retry logic, while model performance was refined through hyperparameter tuning using GridSearchCV [3].

Building on prior research that highlights the role of machine learning in metadata analysis [3] and the importance of feature selection in predicting game success [4], this study provides developers with actionable insights. Our findings aim to empower indie developers to make informed decisions on pricing, release timing, and gameplay features, ultimately helping them stand out in the fiercely competitive gaming market.

0.1 Research Goals

The primary goals of this study are as follows:

- To identify which game metadata features, such as price, recommendations, and genres, are significantly correlated with the success of an indie game [4].
- 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.

1 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.

1.1 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.

1.2 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.

1.3 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 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.

1.4 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.

2 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.

2.1 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.

```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.

```python
# 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.

```python
# 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.

```python
# 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.

```python
# 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.

2.2 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.

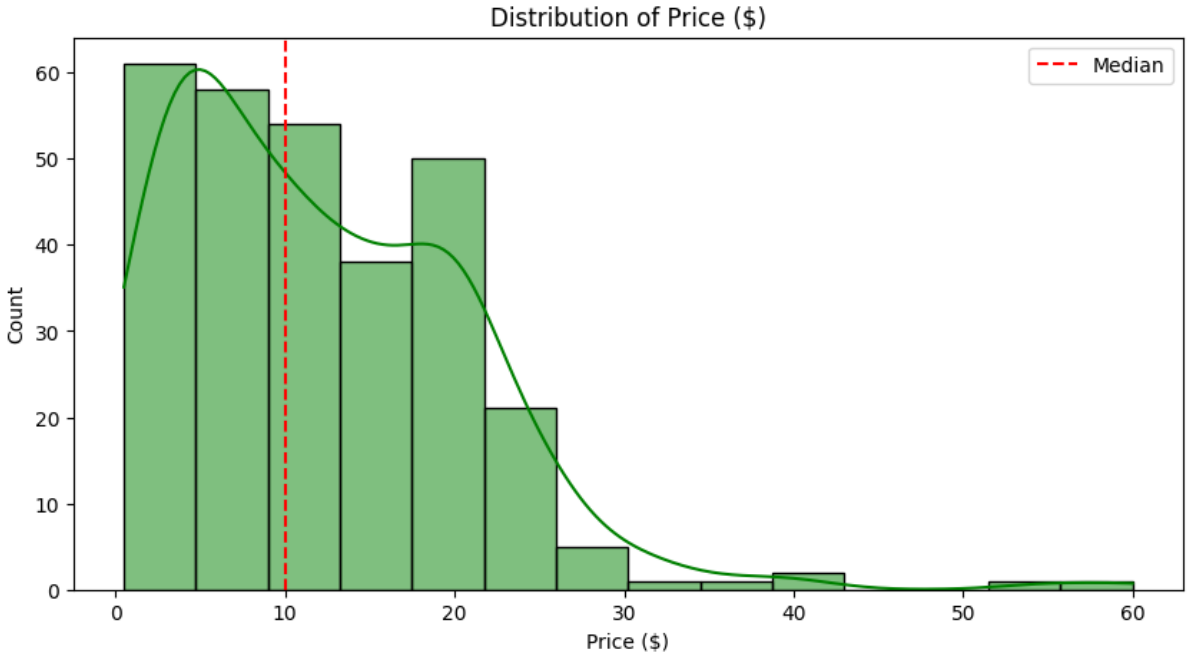

Fig. 9: Distribution of Game Prices. The distribution reveals that most indie games are priced below $20, with some outliers at higher price points. This suggests that indie developers generally adopt a low-price strategy to attract a wider audience. However, the presence of outliers indicates that some 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.

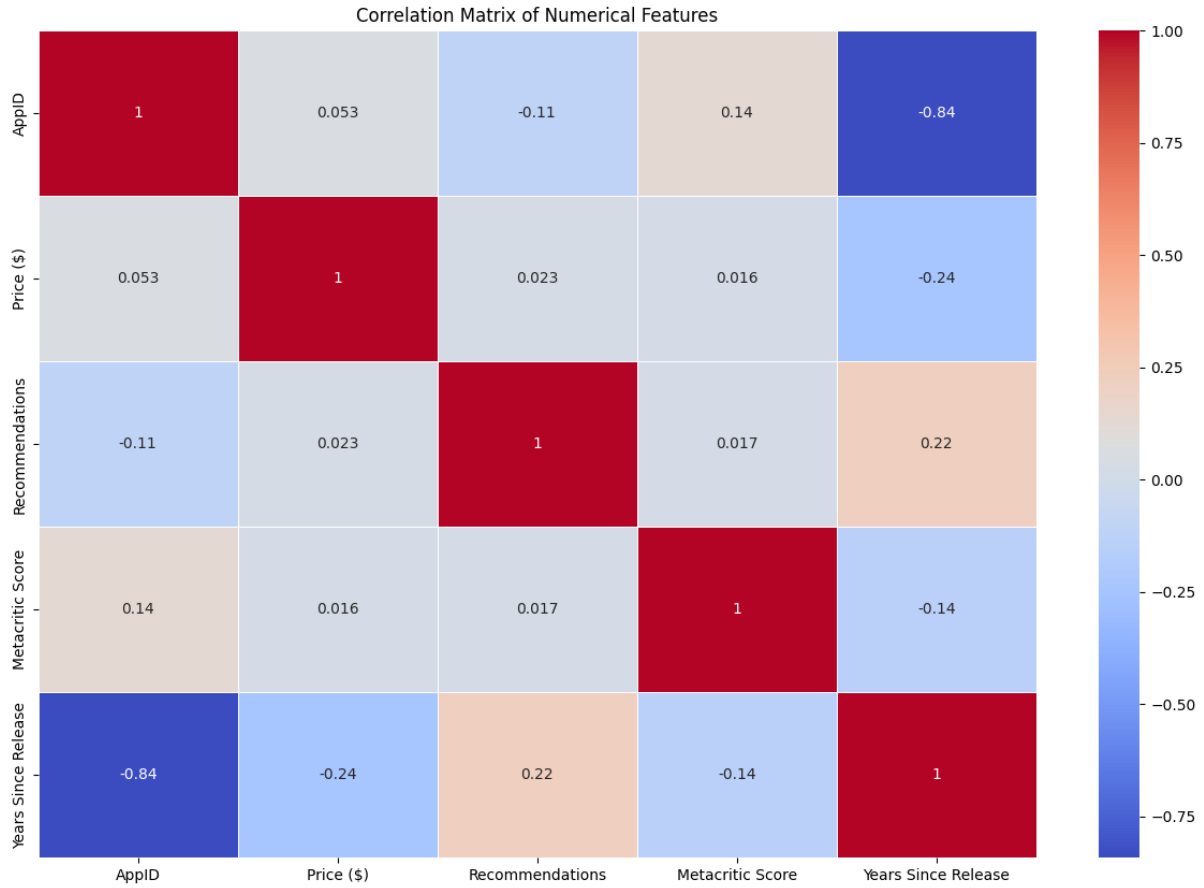

Fig. 10: Correlation Matrix of Numerical Features. The heatmap reveals that most correlations between features are weak, with a notable strong negative correlation between 'Years Since Release' and 'AppID', likely due to the way that Steam assigns ID numbers over time. It also shows that price and recommendations 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.

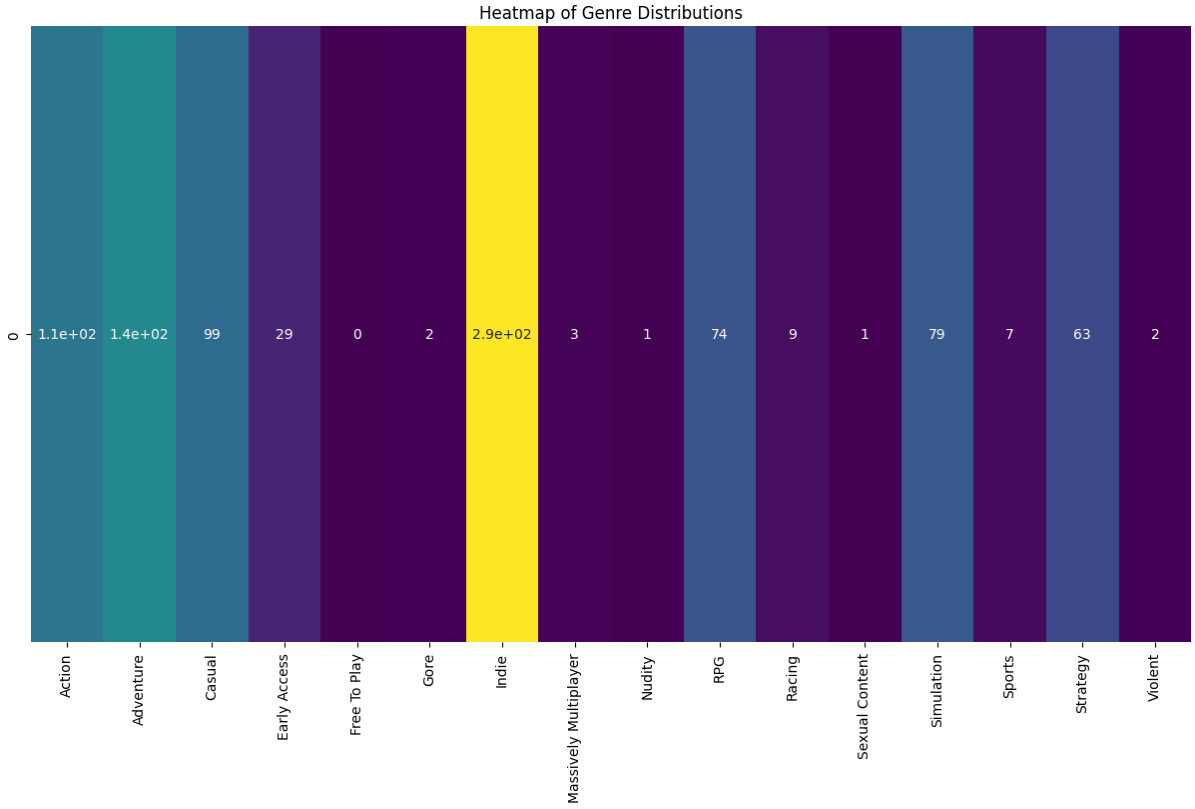

Fig. 11: Heatmap of Genre Distributions. This plot helps in understanding the diversity and commonality of genres within the dataset. Action and role-playing games are the most frequent genres, which indicates their dominance in the indie market. This dominance may be due to the popularity of these genres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

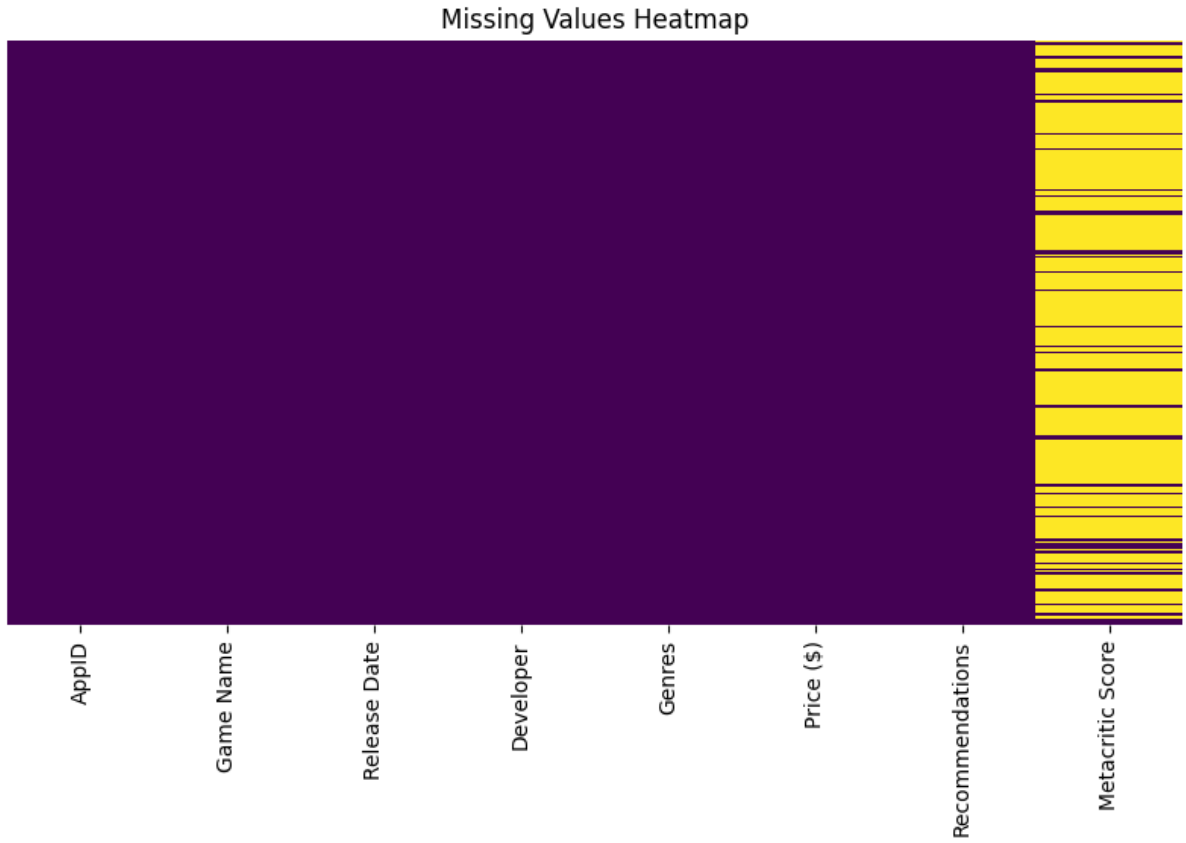

Fig. 12: Missing Values Heatmap. This plot reveals a high concentration of missing values in the 'Metacritic Score' column, while other columns such as 'AppID' and 'Game Name' are mostly complete. Identifying the extent of missing data was crucial for determining appropriate imputation strategies, and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

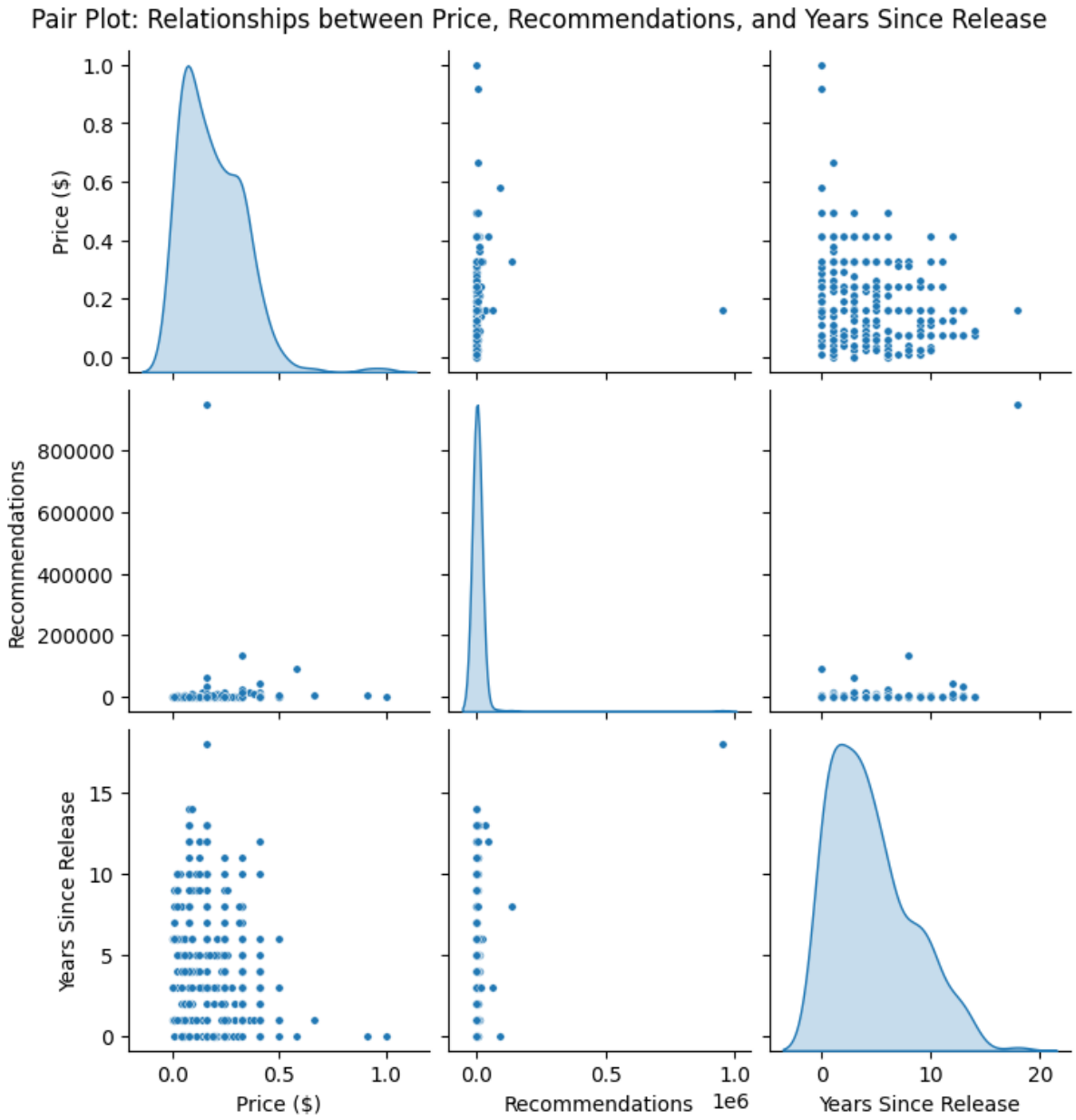

Fig. 13: Pair Plot of Numerical Features. The pair plot provides a comprehensive view of relationships between key numerical features, such as price, recommendations, and years since release. It highlights trends and potential correlations, such as a clustering of low-price games receiving a high number 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.

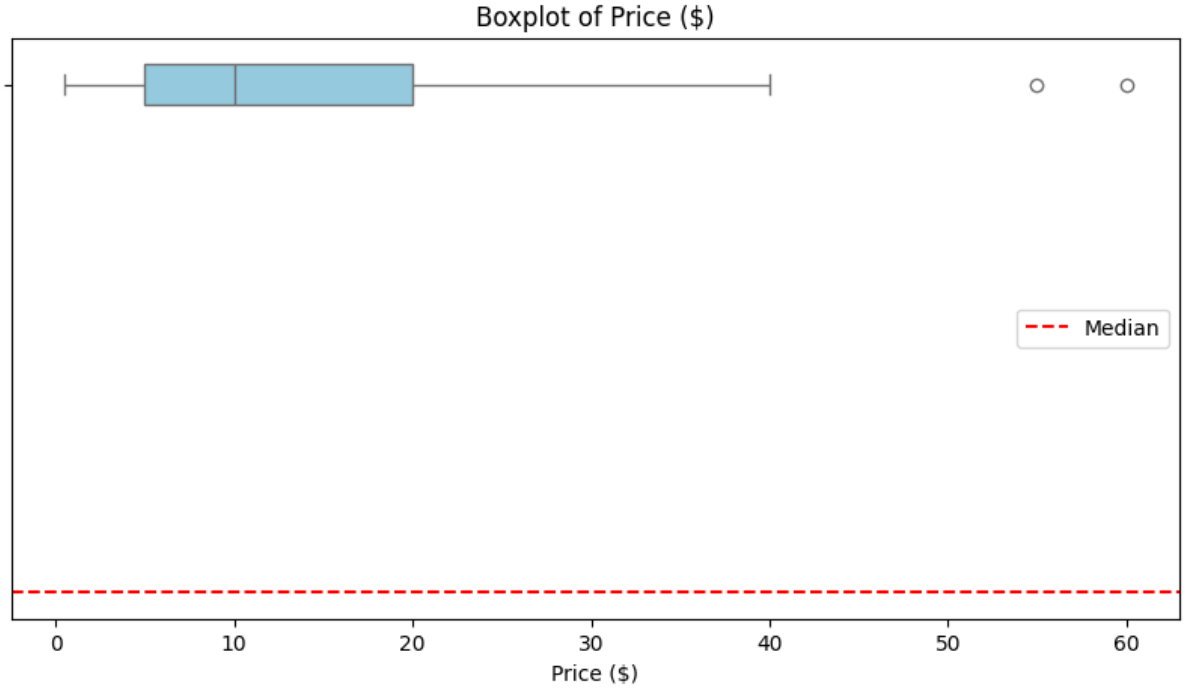

Fig. 14: Boxplot of Game Prices. This visualization displays the distribution of indie game prices on Steam, highlighting that the majority are priced below 20 units. Outliers above 50 units indicate the presence of a few premium-priced games.

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

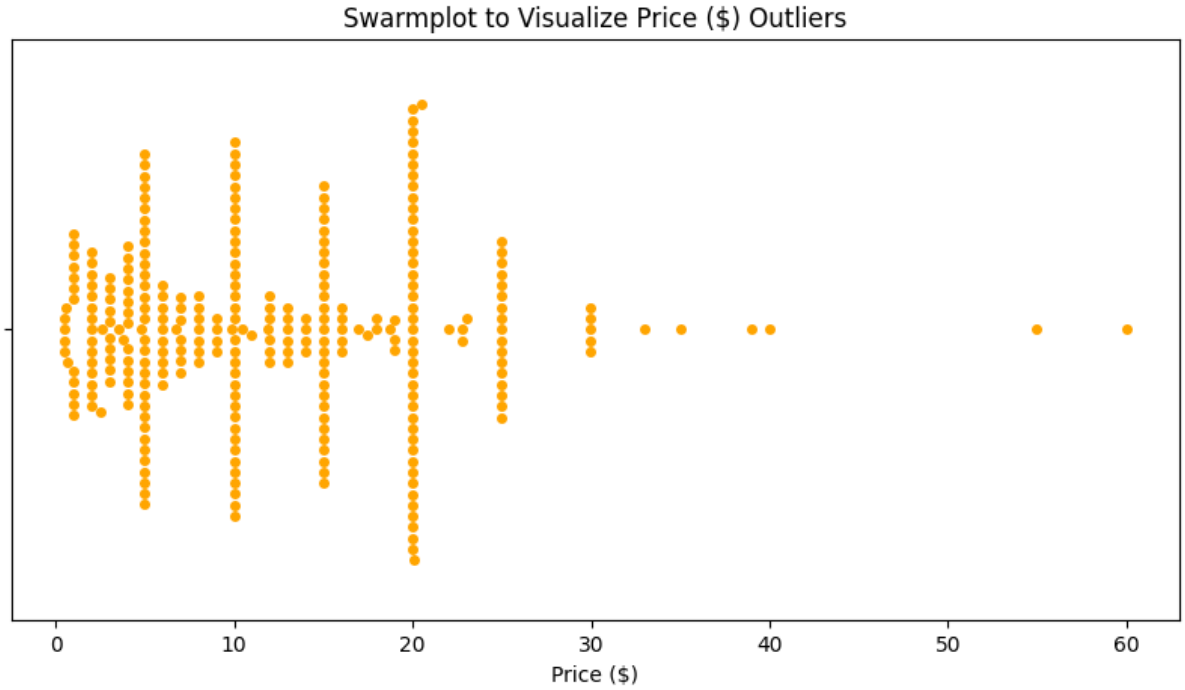

Fig. 15: Swarmplot of Price Outliers. The plot highlights outliers and variability within the game pricing, providing insights into market pricing strategies and identifying unusually priced games.

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

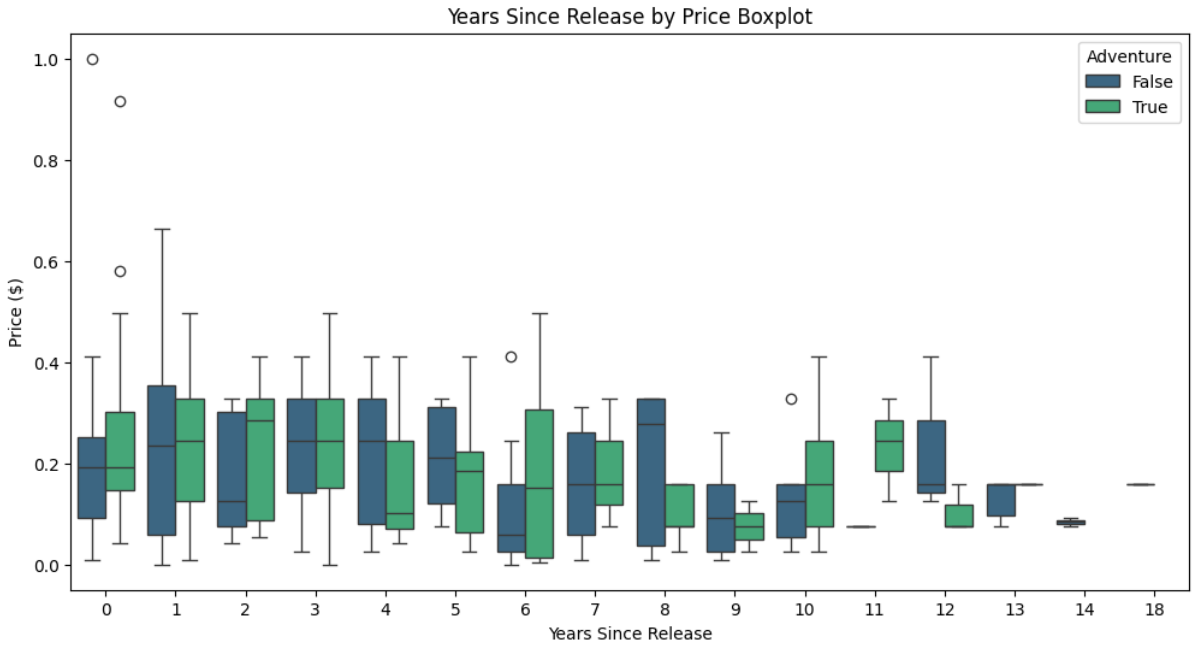

Fig. 16: This plot illustrates the relationship between game prices and the time since release, highlighting that older games generally see lower prices due to discounts or reduced demand. The inclusion of 'Adventure' games reveals slight variations in pricing trends, with these games occasionally retaining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

2.3 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 weak positive correlation was observed between `Years Since Release` and `Recommendations`. This indicates that older games may have had more opportunities to accumulate recommendations, but the relationship is not strong enough to be predictive on its own.
- No significant correlation was observed between `Price` and `Recommendations`, confirming that price alone does not strongly influence player feedback or popularity.

Genre and Feature Relationships

- Weak correlations were identified between game genres and key features such as `Price` or `Recommendations`. This suggests that while genre contributes to the identity of a game, other factors such as gameplay quality, marketing, and developer reputation play a more substantial role in success.
- Some genres, such as `Action` and `Adventure`, appeared more frequently in the dataset, reflecting their popularity among indie developers, though this does not directly translate to higher recommendation counts.

Pricing Trends and Outliers

- Boxplots and swarmplots revealed that the majority of indie games are priced below $20, reflecting affordability as a key strategy for appealing to a larger audience.

– Over time, game prices tend to decrease, as seen in the boxplot of `Years Since Release` versus `Price`. This aligns with industry practices of discounting older titles to maintain sales and consumer interest.
– Outliers in pricing reflect a mix of strategies, including premium pricing for exclusive or high-value titles, as well as promotional pricing for limited-time discounts or bundles.

Recommendations Distribution and Outliers

– The distribution of `Recommendations` was highly skewed, with most games receiving relatively low counts, while a few popular titles garnered disproportionately high recommendations. A log transformation was applied to stabilize variance and improve interpretability during modeling.
– Outliers in `Recommendations` suggest that a small number of highly successful games dominate the platform, likely driven by exceptional quality, strong marketing, or a loyal player base.

Key Challenges Identified

– Class imbalance was evident in the dataset, with significantly more games falling into the low-recommendation category compared to high-recommendation games. This imbalance informed the decision to employ balancing techniques such as stratified sampling during data preparation.
– Missing data, particularly in fields like `Metacritic Score`, posed a challenge, leading to the exclusion of this feature in particular when revisiting the cleaning phase.

3 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 [1], and SVM for flexibility with kernels and handling class imbalances.

3.1 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`).

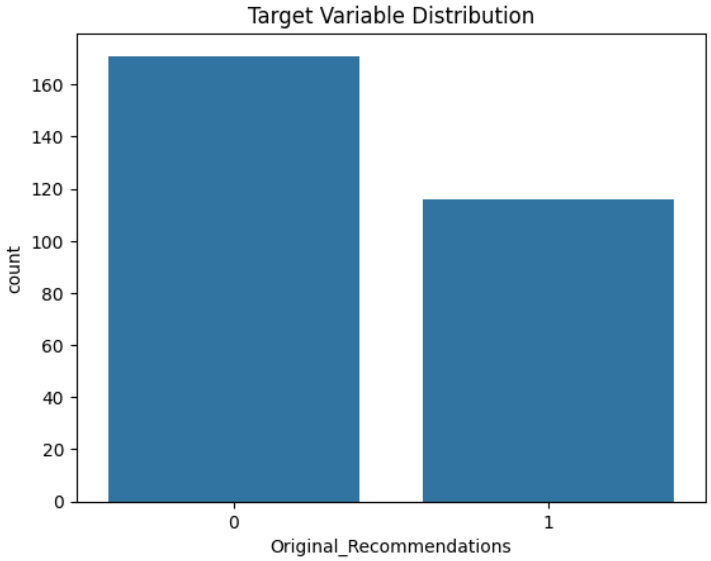

Fig. 17: Distribution of the Binary Target Variable. The chart illustrates the class imbalance, with more samples classified as non-successful (`0`) than successful (`1`). Stratified sampling was used to maintain this distribution in training and testing sets.

3.2 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%

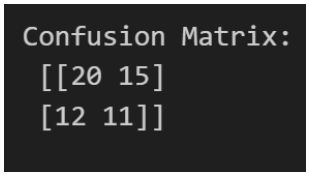

Fig. 18: Confusion Matrix for Logistic Regression (Years Since Release only).

Using Combined Features:

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

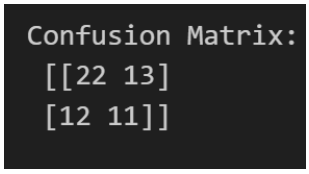

Fig. 19: Confusion Matrix for Logistic Regression (Combined Features).

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%

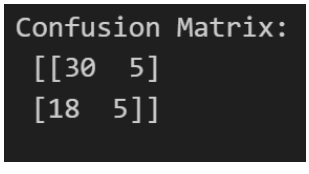

Fig. 20: Confusion Matrix for Random Forest (Years Since Release only).

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).

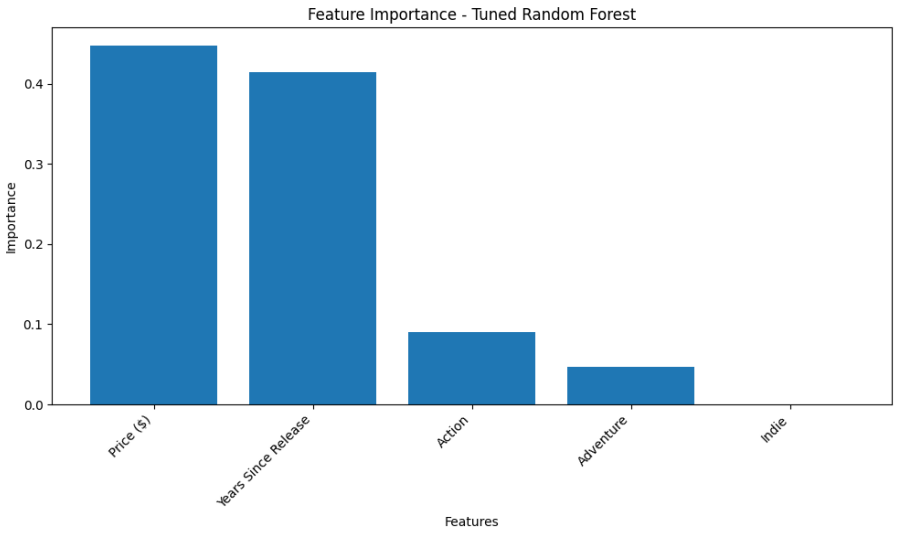

Fig. 23: Feature Importance for Tuned Random Forest (Combined Features). This visualization highlights that `Price` and `Years Since Release` were the most influential features in predicting game recommendations.

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).

3.3 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 has the best performance, using the `GridSearchCV` provided by Scikit-learn [5]. 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.

3.4 Model Comparison and Key Findings

Table 1 summarizes the performance of the evaluated models on the test dataset, providing accuracy, precision, recall, and F1-score metrics for both the majority class (Class 0 - non-successful games) and the minority class (Class 1 - successful games). This comparison highlights the effectiveness of each model in handling the class imbalance and predicting game success based on different feature sets and configurations.

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 (Untuned)	Combined Features	60.3%	66%	71%	50%	43%	58%
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 (Untuned)	Combined Features	50.0%	58%	60%	36%	35%	47%
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 with this particular dataset.

- **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 [2].

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.

4 Conclusions and Future Work

Developing a successful indie game on a competitive platform like Steam can be daunting. This project aimed to provide developers with clear, data-driven insights into what contributes to success and actionable strategies to help their games stand out. By using machine learning models and analyzing key features like pricing, release timing, and player recommendations, I hope this research offers meaningful guidance for developers and perhaps even some validation for players looking to support quality indie titles.

4.1 Key Findings

The findings from this project highlight important trends that developers can use to refine their strategies:

- **Machine Learning Performance:** Among the models tested, the tuned Random Forest performed the best with an accuracy of 65.5%. This model pinpointed two critical factors: `Price` and `Years Since Release`, suggesting these are pivotal in determining game success on Steam (Figure 23).
- **Affordable Pricing Wins:** Games priced below $20 were consistently more successful, showing that affordability is a significant factor for many players. The price distribution plot (Figure 9) highlights this trend while also showing outliers priced above $50, reflecting premium strategies.
- **The Long Game Matters:** Older games tended to have more recommendations, indicating that maintaining relevance through updates, community engagement, or sales campaigns over time can have lasting benefits. This relationship is illustrated in the pair plot of numerical features (Figure 13).
- **Genre Popularity Doesn't Guarantee Success:** Action and adventure genres dominated the dataset, but their frequency didn't always translate to higher recommendations. The heatmap of genre distributions (Figure 11) suggests that developers may find opportunities in underserved niches while ensuring their gameplay delivers a unique, engaging experience.
- **Challenges with Class Imbalance:** The dataset contained significantly more games with low recommendations (Class 0) compared to high recommendations (Class 1), as illustrated in Figure 17. This imbalance posed challenges for the models, particularly in predicting success for games with fewer recommendations. Stratified sampling was used to maintain this distribution in training and testing sets, but further techniques like oversampling could improve results.

4.2 Actionable Insights for Developers

These findings offer some practical strategies for indie developers:

- **Affordable Pricing:** Consider pricing your game below $20 to make it accessible to a broader audience. While premium pricing can work, it requires the product to have exceptional gameplay, visuals, or a strong marketing strategy.
- **Long-Term Engagement:** Successful games often take time to gain traction. Frequent updates, bonus content, or seasonal events can keep your game relevant and encourage player recommendations.

- **Exploring Niches:** While popular genres dominate the platform, don't be afraid to explore unique ideas. Niche genres with well-executed mechanics and a clear value proposition can carve out a loyal player base.
- **Quality First:** Players value high-quality gameplay and experiences over pricing alone. Focus on delivering a polished product that resonates with your audience.

4.3 A Word for Players

This study isn't just for developers—it's also for players who love indie games. As a player, your choices shape the market. When you support quality indie titles, especially those in niche genres, you're helping to grow a vibrant and diverse gaming ecosystem. Look beyond surface metrics like genre or price and dive into hidden gems that may offer truly unique experiences.

4.4 Limitations and Challenges

While this project provided valuable insights, there were some challenges:

- **Class Imbalance:** The dataset contained significantly more games with low recommendations, making it harder to predict the success of underrepresented games with higher recommendations, as seen in Figure 17.
- **Data Gaps:** Missing values in key features like `Metacritic Score` limited the completeness of the analysis. This is a reminder that even with powerful tools, the quality of data plays a significant role in determining outcomes.
- **Platform Specificity:** This research focused solely on Steam, which means its findings might not generalize to other platforms like Epic Games Store or console marketplaces.

4.5 Future Work

Moving forward, there are several opportunities to expand upon this project:

- **Improving Predictions for Niche Titles:** Exploring advanced techniques like SMOTE (Synthetic Minority Over-sampling Technique) could help better predict success for less popular games.
- **Expanding Metadata:** Adding features like user review sentiment or playtime could offer richer insights into what makes a game successful.
- **Multi-Platform Analysis:** Including data from platforms like Epic Games Store, Itch.io, or even consoles would make the analysis more comprehensive and applicable across the industry.
- **Seasonal Trends:** Investigating the influence of release timing, such as launching during holiday seasons or alongside major sales, could yield actionable insights for developers.

In conclusion, this project represents a first step toward empowering indie developers with data-driven strategies. By understanding and applying these insights, developers can better navigate the challenges of the gaming market, while players can discover and support games that truly resonate with them based on their personal preferences. Whether you're building the next hit game or just looking for your next favorite title, there's something here for everyone.

Additional Resources

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

- Overleaf Report
- GitHub Repository
- Data Cleaning and EDA Notebook
- Predictive Analysis Notebook
- Steam Data Collection Script
- GitHub README Documentation
- Results and Reports Directory
- Visualizations and Supporting Diagrams
- Project Requirements File
- Steam API Documentation

References

1. Breiman, L.: Random forests. Machine Learning **45**(1), 5–32 (2001). https://doi.org/10.1023/A:1010933404324, https://doi.org/10.1023/A:1010933404324
2. Chawla, N.V., Bowyer, K.W., Hall, L.O., Kegelmeyer, W.P.: Smote: Synthetic minority over-sampling technique. Journal of Artificial Intelligence Research **16**, 321–357 (2002). https://doi.org/10.1613/jair.953, https://doi.org/10.1613/jair.953
3. 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
4. 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
5. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: Machine learning in python. Journal of Machine Learning Research **12**, 2825–2830 (2011), https://jmlr.org/papers/v12/pedregosa11a.html