Influences on Game Success on Steam

GROUP 14
Andy Salazar Parra

Contents

Abstract

Steam is a popular game distribution platform for computer games. The games in its vast library are highly varied in terms of their features, like game mechanics, genres, system requirements, etc. Knowing which features can influence the success of a game in terms of the number of how many people own the game, or in terms of the proportion of positive reviews, could be useful from a marketing perspective. To answer these questions Decision Tree Classifiers were used. In a model with 74.5% accuracy, the number of owners was influenced mostly by the proportion of recommendations, but also by whether the game is free to play, the proportion of positive reviews and whether the game was multiplayer. Unfortunately, the Decision Tree Classifier for the proportion of positive ratings had a very low accuracy and can thus offer no meaningful insight to what features influence this success measure.

Influences on Game Success on Steam

Background and Motivation

Steam is the most popular game distribution platform and store front for computer games. It is owned by Valve Corporation, a game developing and publishing company. Steam was founded in 2003 and was at first limited to the distribution of Valve games. It then expanded to distribute third party games in 2005. Small or independent developers can publish their games directly on Steam through the use of the Steamworks API made freely available in 2008 (Wilde & Sayer, 2022). There are almost 100,000 games on the platform, ranging from AAA games like the Assassins' Creed series, developed by large multinational studios, to indie games like Stardew Valley, developed by a single person. These games encompass a large variety of genres, play styles, prices, controller and VR support options, languages, system requirements, etc. So, can we use data mining to predict which games will be successful, and find out which features influence that success? Are, for example, First Person Shooters naturally more popular? Is the support of Mac OS in high demand? The answer to these questions could help developers, publishers, and investors make informed estimates about the success of their game or help them choose which aspects of a game to prioritize.

As for what quantifies success, we can look at two aspects: How many owners does the game have on steam, which reflects a certain financial success, and how positively rated the game is, which reflects success in terms of public reception.

One way to answer these questions is by using decision tree classifiers, to classify games in terms of the number of owners, and the proportion of positive reviews, and identifying which features have higher predictive power in the models created.

To the best of my knowledge this research is entirely new. While there has been some analysis done on Steam game data, this has been mostly confined to exploratory data analysis focusing on describing the data, as well as the creation of recommender systems.

Problem Statement

To summarize, my main research question is: What features of a game influence it's success in terms of its number of owners, and its proportion of positive reviews.

Owners in the Steam Store refers to the number of people who have played the game using the Steam platform. It can include purchases made outside the Steam store if the game is activated on Steam and also games received as a gift. Games that are free to play are also included as long as the user opens the game, and this can sometimes lead to an inflation in the number of owners if the game is made free for a

limited time. The owner status for these limited time offers is removed once the offer ends if the user does not purchase it (Gaylonki, S. n.d.).

Method

Data

Because I enjoy coding and wanted to try something new and challenging for this project, I wanted to do it fully on Python, beginning with data extraction. So, for this project I created my own data set by extracting data from the Steam Web API, as well as the Steam Spy API.

The Steam web API provides all the data about the game itself, but unless one is a developer for a specific game, there is not much user related data available due to data privacy laws. The Steam Spy API uses the Steam API to provide an estimate of the range of the number of owners a game has. It is unclear exactly how this estimate is obtained, as the owner of the site is protective of the code and raw data he uses, but it is accepted as a generally accurate estimate, since it provides a range that takes into account a wide margin of error, and not an exact number. The Steam SPY API is also useful for extracting the number of positive reviews (Gaylonki, S., n.d.).

Data Mining Method

The main data mining function I employed was decision tree classification. I did it on Python using the DecisionTreeClassifier from the Scikit-Learn library. The algorithm employed by Scikit-Learn is an optimized version of the CART algorithm (Scikit-Learn, n.d.). I used the entropy criterion for splitting the tree because this is the formula we studied in class, so I have the most knowledge of how it works.

It is also because I have more familiarity with this method over others, that when I was creating the classifier for the model that would predict the proportion of positive reviews, I chose to split the continuous rating values into categories, as opposed to using a Decision Tree Regressor.

Data Extraction and Merging

Data was extracted using the request module on Python (See dataExtraction.ipynb).

First, I wrote a function to handle requests that takes the API URL and parameters as input and uses try-except, to handle errors. The initial extraction was done all at once and returned all IDs for the applications on the Steam Library along with their names. The response was decoded in JSON format, and the AppID and names were retrieved from the JSON structure. They were saved as columns in a Pandas data frame and then exported as a CSV file (app_list.csv). In this initial data set, there were 191,906 rows.

I then stored the AppIDs form that data frame in a python list. By iterating over this list, the IDs were passed as parameters in another function that gets each AppID and passes it as a parameter in a request to the Steam API. The data from the responses was then processed as a Pandas data frame and later exported as a CSV. The extraction process was done in batches of 100 IDs at a time. It took several days to complete the extraction for all IDs in the initial AppID dataset and the resulting data set had 44 columns, and 176,389 rows where the Steam_AppID column was not null.

Then I used Pandas functions to drop the rows that contained duplicated AppIDs. In the App Data data frame, there was a column called type, which denoted the type of application and included not just games, but DLCs, demos, etc. Because my interest was in video game features, I decided to drop all rows where

the “Type” column did not contain the value “Game.” This lowered my number of rows to 105,790 (steam_app_data.csv).

Using the remaining AppIDs, I extracted the data associated with these IDs from the SteamSpy API, using the same process as I used to extract data from the Steam Web API. The resulting data set had 20 columns and 105,790 rows.

Merging this data set was a simple process as they had matching AppID columns, so I used the pandas merge function joining on those columns, resulting in a data frame with 64 columns and 105,790 rows.

I dropped some columns that where redundant because there was some overlap in the data extracted from Steam and the one extracted from Steam Spy. Other columns were highly dependent on the time the data was extracted, so because the values fluctuated so much, I dropped them. Some other columns I decided to drop because they did not seem relevant.

***Dropped Columns*:**

- **name_y:** This duplicated the information from the “name_x” column.
- **appid:** This duplicated the information in the “steam_appid” column.
- **developer:** This duplicated the information in the column “developers.”
- **publisher:** This duplicated the information in the column “publishers.”
- **languages:** This column from Steam Spy contained less complete language information than the “supported_languages” column from the Steam API.
- **genre:** This column from the Steam Spy API also contained less complete genre information than the equivalent “genres” column from the Steam API.
- **price_overview:** This column from the Steam API contained a dictionary that stored the information found in the “price”, “initial price” and “discount” columns from the Steam Spy API. I decided to eliminate this as the other columns had already separated this information.
- **alternate_appid:** This column contained ids for related applications like DLCs. It did not seem relevant.
- **ccu**: This stands for concurrent users and reflects the peak number of players for the day prior to the day the data was extracted on. Because this value is so dependent on the time of extraction, I decided to drop it.
- **score_rank:** This ranks games based on reviews, but it was null for most rows. Since I already had the number of positive and negative reviews for each game I dropped it.
- **fullgame:** This column had information on the full game for applications that were DLCs. Because I had dropped apps that were not games this was null, so I dropped it.
- **average_2weeks:** This column reflects the average playtime for this app in the last 2 weeks. Because it was dependent on the time of extraction, I dropped it.
- **median_2weeks:** similarly, this column had the median playtime for the last 2 weeks and it was dopped for the same reason.
- **price:** This column was the current price of the game at the time of extraction. Because this is also time dependent, I dropped it.
- **discount:** This column had discounts applied to the game at the time of extraction. I dropped it for the same reason.
- **userscore:** This column had a 0 value for all, but 44 rows and I could not find any information about it in the Steam API documentation, so I dropped it.

After this process I was left with a data set that contained 105790 rows and 48 columns.

Data Cleaning and Preprocessing

Initial Cleaning and Preprocessing

The raw data was not very useful for analysis because there were multiple columns of large text strings, sometimes including html tags, or lists and dictionaries containing multiple values. So, it required a lot of cleaning and preprocessing. Appendix A provides a summary table of this process. In the interest of staying within the page limit I will discuss the process broadly, providing more details for the more challenging transformations; for the exact code see "dataCleaning.ipynb".

For most of the text data that would be unique to each game, such as the name and description, I simply created Boolean columns that reflected whether the game had that attribute or not.

Some other string columns, like the supported_languages one, were more interesting as the languages supported by each game could affect its marketability. I wrote a function to clean that data that uses the html parser from the Beautiful Soup library. I also used the regular expression module to process any remaining html tags and unnecessary text inside brackets. This column was particularly challenging because the languages themselves were written in different languages and there was no associated language id. To handle this, I took each unique string in this column and manually created a dictionary with the language mappings. I made the English spelling for each unique language a dictionary key and all its translations and regional variations were added as values. I used the keys to create a column for each language and the values to compare them to the language list in each row of the original data. If the language was present, the value for that language column would be True, otherwise it would be False.

The columns for PC, Mac, and Linux requirements were also of interest. These columns contained a dictionary of minimum and recommended requirements related to OS, processor, memory, graphics, and storage. But these dictionaries were quite messy. Some had html tags, and some had the minimum requirements inside the recommended requirements or vice-versa. So, instead of accessing the values from the dictionary structure I cleaned it in a similar manner to the supported languages column by manually creating a separate dictionary. These three columns were usually identical except for the OS requirements, so decided to ignore the OS requirements and use the processor, memory, graphics, and storage requirements from the pc_requirements column (which all games had values for). I created new columns for all these four aspects. This time I created three separate dictionaries, one for each column, with keys that indicated high, medium, low requirements and the values were specs relating to those categories. For example, in the processor dictionary the key "low" had values like: i3, ryzen3, and dual-core. I built this dictionary after inspecting the data for common values, but I was less thorough than with the languages column as there were many more unique values to handle, so I relied more on my own knowledge of pc requirements to create the dictionary as opposed to mapping every single unique value.

The developers and publishers columns contained lists of string values, so I used the literal_eval function from the ast module to get the program to read it as a list, then I examined how many rows had more than one value in this list. Because most had only one, I decided to only keep the first value in these columns.

Using literal_eval function helped process many columns that already had structured data inside dictionaries. This was especially useful for processing the genres and categories columns. While they contained categories and genres spelled in different languages, they were all already mapped to distinct ids, so it was just a matter of creating a new dictionary with the id and the English label for that category, then checking it against the data through a similar process as the one used for the languages requirement columns.

The release date column had the release date as well as a Boolean “coming soon” value stored in a dictionary. I split them into separate columns and then made the release date a datetime value.

The ratings dictionary had some information regarding age requirements and content warnings across rating systems. This information was already contained in the required_age and has_content_warning columns, however I noticed that many rows that had null or 0 values for these columns had some other information in the ratings column. My solution was to search through the different rating system dictionaries to get all the required ages, then take the average age, and use that to update the required age column for those that had missing values. Then I looked at the descriptors key within the ratings dictionaries and used that to update the has_content_warnings columns too.

Because I wanted a measure of positive and negative reviews that would not be as dependent on the game popularity, I took the positive and negative reviews columns and used them to calculate the total number of reviews and the proportion of positive as well as the proportion of negative reviews. Since recommendations are only done when reviewing the game, I took a similar approach to calculate the proportion of recommendations based on the total recommendations and the total user reviews.

I also decided to transform the average and median playtime columns which had values in minutes to hours and the initial price column from cents to USD, so that the values could be more meaningful at a glance.

The controller support column was eliminated because that information was already contained in the columns for the categories full_controller_support and partial_controller_support.

At the end of this initial process there were 105,790 rows and 655 columns.

Feature Reduction, Missing Value and Outlier Handling

Because over 600 columns seemed excessive, I decided to drop the Boolean columns for which less than 5% of the data had a True value. The rationalization was that if only a few games had this feature, it would probably not add much to the model in terms of predictive power. As I expected, the majority of features dropped were uncommon languages, genres, categories, and tags.

The processor, memory, graphics, storage, developers, publishers, release date, and initial price columns had missing values. Most of the rows that were missing a release date had a True value for the column “coming_soon”, meaning the game was unreleased, so I dropped all of the rows that did not have a release date. Most of the rows that did not have an initial price had a True value for the “is_free” column, so I made the ones where this was true have an initial price of 0. The ones that were not free but were still missing an initial price were dropped. For developers and publishers at least one of these values was present in most rows, so for the ones that were missing one but not both, I duplicated the value. That is to say, if a game had Riot Games as a publisher but a missing value for developer, I put Riot Games as the developer too. Publishers and developers are different things, but they are closely related features and sometimes both functions are conducted by the same company, so this approach seemed most appropriate. The rows with missing values in both of these columns were dropped.

For the processor, memory, graphics, and storage categories, I expected that there would be many missing values considering the less thorough approach I took when making these columns. Luckily most rows had values for at least one of these columns, so I created a general systems requirement column and wrote a function that would take the highest value in these four columns and input that as the system requirement value. For example, if a game had a value of “high” in memory, storage was missing, graphics were “medium”, and processor was “low,” the new system requirements column would have a value of “high.”

For the rows where all four columns were missing, since it was still over 10,000 rows, rather than dropping them I replaced the null values with “unknown.”

Some rows had impossible required ages like 1000. If those games had a content warning, I gave them the maximum required age of 21, and if they didn’t, I gave them a required age of 0.

After this process I had 80,116 rows and a more manageable 129 columns.

Preparing Data for use in Decision Tree

For use in the decision tree model, I needed to convert all data into numbers. I turned the unique values in owners, publishers, developers into categorical data, and then created dictionaries to map the categories to codes and then used the mapping function to create new columns with the category numbers.

For the release date data, I split it into release year, release date of month, release day of week, and release month. While I use datetime functions to get most as numerical data, for release day of week, I first obtained the name of the day (i.e. Monday), then made this a categorical value and manually mapped it to numbers. I took this extra step because it was easier for me to understand the days of the week if they were represented by numbers 1-7 instead of 0-6.

In the categories for the owner data, only two games had more than 100,000,000 owners so I merged those two with 9 that had the category of 50,000,000 – 100,000,000 owners by creating a 50,000,000+ category.

The proportion of positive reviews needed to be turned into categorical data too, for use as a class attribute in a decision tree classifier, so I used the pandas cut function to create bins for the proportions, and then gave these bins a numerical label.

After this process I had 80,116 rows and 136 columns, now with the added category code columns.

Decision Tree – Number of Owners

Initial Tree

To create the model, I first imported the necessary libraries (descisionTreeModels.ipynb). Form Scikit-Learn in particular, I imported the decision tree classifier, as well as the related plot_tree and export_text functions. I also imported train_test split, stratified K fold, GirdSearchCV, and the CrossValScore form their model selection module. As well as accuracy_score, from their metrics module.

I then created the x data frame with the features I selected for use in the decision tree. Because I was interested in exploring the possible impact of most game features, I wasn’t very selective. I dropped all object type columns (they already had corresponding numeric columns). Then I dropped “owners_code", which was the class column, "total_recommendations", "total_user_reviews", because I felt these values would be closely related to the number of owners class and were probably used to estimate the number of owners in the first place, "median_playtime", "average_playtime", because I felt these would also reflect ownership given that games with more owners are bound to have more people that pay it for longer. And "reviews_proportion_negative”, and “reviews_proportion_positive_bin_code” because these values were already represented in the data set by the reviews_proportion positive column. The reviews_proportion_ negative column is just the complement to the reviews_proportion_positive column, and the reviews_proportion_positive_bin_code is just the category codes for the proportion of positive reviews.

I then created the y dataset with the class column, which was “owners_code.”

Table 1. Each class label with it's associated numerical code and the number of games that had that class value.

Numeric Code	Owners Class Label	Frequency Count
1	0-20,000	58,433
2	20,000-50,000	9,457
3	50,000-100,000	4,535
4	100,000-200,000	2,999
5	200,000-500,000	2,432
6	500,000-1,000,000	1,078
7	1,000,000-2,000,000	610
8	2,000,000-5,000,000	373
9	5,000,000-10,000,000	120
10	10,000,000-20,000,000	44
11	20,000,000-50,000,000	24
12	50,000,000+	11

I then split the x and y data frames into training and testing data using train_test split. I used a test size of 0.3 and I did stratified sampling on y, because as can be seen in Table 1 the data was positively skewed with most games having fewer owners and very few games having a higher number of owners. So, I wanted to ensure an approximately equal distribution of class values in the training and testing data. I also used random_state = 1 for reproducibility of the split.

I then created a decision tree model using the entropy criterion, and with random_state=1 for reproducibility. Otherwise, I used the default hyperparameters (max-depth: None, min_samles_split: 2, min_samples_leaf: 1) (Scikit-Learn, n.d.). I fitted this model to the training data.

I used the export_text function to obtain the txt file with the decision tree (see "decision_tree_owners.txt") and I plotted it using the plot_tree function and the matplotlib library.I'm not including the tree plot, because the tree was so large the image is unreadable.

Results. I used Scikit-Learn's feature_importances_ property to get the importance of each feature on the model's predictive power as determined by its contribution to entropy reduction. I sorted the features by importance in descending order to get the ones with highest importance first, I plotted them using matplotlib. The full graph can be seen in "featureImportancesOwners.png." It was too large to be included here. I then printed the most important features and their importance score.

Table 2. Top 5 Features and their Importance for Number of Owners Tree

Feature	Importance
proportion_recommended	0.186
reviews_proportion_positive	0.083
release_year	0.046
publishers_code	0.045
release_day_of_month	0.044

Then I calculated the accuracy of the decision tree classifier by using the predict function for the testing features, and compared them to the actual testing class values, using the accuracy_score function.

The accuracy was 0.6835, meaning it correctly predicted the classes only 68.35% of the time. To validate this accuracy and ensure it wasn't just an artifact of how the data was split, I used the cross_validation_score feature using the stratifiedKfold method with 5 splits, calculating the accuracy of the model with 5 different configurations of training and testing data. The mean accuracy score after cross validation was 0.6801 with a standard deviation of 0.002.

This accuracy score was a bit lower than I would've liked, so I decided to try pruning the tree to reduce the risk of overfitting and improve the accuracy.

Pruned Tree

To prune the tree, I used Scikit-Learn's GridSearchCV function. This let me create a dictionary of different configurations of hyperparameters to be used in the tree. I then fitted the model using each combination of the grid parameters and used a 5-fold cross validation for the performance of each combination.

I then used the best_param attribute of GridSearchCV to get the best parameters, which were max_deph = 5, min_samples_leaf = 5, and min_samples_split = 2.

I created the pruned decision tree model using these parameters, and once more fitted it on the data. I exported it as text (decision_tree_owners_pruned.txt) and plotted it. While the plot is still too large to be included in this document, this time it was readable enough that it is included in the project folder (decissionTreeOwners_Pruned.png).

Results. I again used the feature_importances_ property to get the features with the most predictive power. I plotted them (featureImportancesOwnersPruned.png) and printed the top features along with their importances.

Table 3. Top 5 Features and their Importance for Pruned Number of Owners Tree

Feature	Importance
proportion_recommended	0.588
g_ftp	0.106
reviews_proportion_positive	0.091
tag_Multiplayer	0.051
release_year	0.045

I then calculated the accuracy of the model using the testing data. The accuracy was 0.7452, meaning the model made accurate predictions about 74.52% of the time. Then I got a stratified 5-fold cross validation score for this accuracy and got a mean accuracy of 0.7454 with a standard deviation of 0.001.

Discussion

The pruned tree had a better accuracy, so to find out which features of a game influence the number of owners, it makes more sense to look at the feature importances for this model. The features that seemed to influence the prediction of number of owners were: how many out of the people who reviewed it also recommended it, whether the game had the free to play genre, how many of the reviews it had were positive, whether users tagged it as multiplayer, and the year it was released. None of these features had a perfectly pure split where I would be able to say that the value for that feature on its own is enough to predict the number of owners. When the proportion of recommendations was equal to 0 the class was equal to 1 (0-20,000 owners) with the exception of games that had a proportion of positive reviews higher that 0.07, and the genre free to play. The highest class in the tree was 9, which corresponds to 5,000,000-

10,000,000 owners. It seems that what influences getting that class is having a proportion of recommendations higher than 0.09, being released before 2016, having the tag Multiplayer, being available in Chinese, and having the tag Fist Person Shooter. The second and third highest classes (1,000,000-2,000,000 & 2,000,000-5,000,000 owners) were predicted at least in part by having a proportion of recommendations greater than 0 and having the tag multiplayer. We can conclude from this that players rely on the feedback from other players in terms of recommendations or positive reviews, when deciding whether to "own" the game, but that the game being free to play and multiplayer also has an impact. Free to play games being popular makes sense because it doesn't require an initial financial investment to play the game (which does not necessarily mean the game is free, as a lot of these games offer in-game purchases that are often necessary to progress with the game). Multiplayer games being popular also makes sense taking into account that a lot of the most popular computer games, like Valve's Counter Strike 2, are both free to play and multiplayer, emphasizing competitive cooperative play to encourage players to purchase competition boons.

Decision Tree – Proportion of Positive Reviews

Initial Tree

For this decision tree I used the same process as I did for the number of owners decision tree classifier. The main differences were the features and class selected.

For my features I selected every feature except proportion_reviews_positive_bin_codes, which was the class I was interested in predicting, and the equivalent column, reviews_proprotion_positive which just had the numeric values uncategorized. I also excluded the prorportion_reviews_negative, because again this is just the complement to the positive proportion, and the total recommendations and the proportion of positive recommendations because those seemed like a feature that would be too closely associated to positive reviews.

For my class attribute I used the proportion_of_postive_reviews_bin_code column, which has categories that encompass a range of proportions of positive reviews, which is inclusive of the lower bound and excludes the highest (See Table 4). I created this column because the the class attribute in a decision tree classifier needs to be categorical.

Table 4. Each class label with it's associated numerical code and the number of games that had that class value.

Numeric Code	Proportion of Positive Reviews Class Label	Frequency Count
1	-0.001 - 0.2	14,957
2	0.2 - 0.4	3,550
3	0.4 - 0.6	8,992
4	0.6 - 0.8	18, 302
5	0.8 - 1.001	34,315

I once again split the data with a 70-30 stratified split and fitted the model on the training data. I also plotted the tree and exported it as text (decision_tree_rating.txt).

Results. I plotted the feature importances (featureImportancesRatings.png) and printed the importances for the top features.

Table 5. Top 5 Features and their Importance for Proportion of Positive Reviews Tree

Feature	Importance
total_user_reviews	0.356
developers_code	0.049
release_day_of_month	0.047
publishers_code	0.045
initial_price	0.036

Then I calculated the accuracy of this model by comparing the predicted scores and the actual scores for the test data, and got an accuracy score of 0.5233, which means that this model is only accurate about 52.33% of the time. I then calculated the mean and standard deviation for the accuracy using 5-fold cross validation and got a mean accuracy of 0.5258 with a standard deviation 0f 0.005, which is only slightly above chance level so the decision tree might as well be making random guesses.

To try to improve the model accuracy and reduce the risk of overfitting I used the decissionTreeClassifier hyperparameters to prune the tree.

Pruned Tree

I used the GridSearchCV function again to select the best hyperparameters for this model and got as a result max_depth = 10, min_samples_leaf = 10, min_samples_split = 2.

I plotted the tree and exported it as text (decision_tree_rating_pruned.txt).

Results. I plotted the feature importances (featureImportanceRatingsPruned.png), and I printed the top ones.

Table 5. Top 5 Features and their Importance for Proportion of Positive Reviews Pruned Tree

Feature	Importance
total_user_reviews	0.840
total_achievements	0.033
release_year	0.031
tag_2D	0.023
initial_price	0.006

This model had an accuracy of 0.5861. I then calculated the mean accuracy after using 5-fold cross validation and got a mean accuracy of 0.5908 with a standard deviation of 0.002. Using pruning the accuracy improved a bit but it is still only a little higher than chance level.

Discussion

Once again, the pruned tree had better accuracy, so it is better to look at the feature importance of this tree to understand what game features influence game success in terms of the proportion of positive reviews. It seems that total_user_reviews was much more important that any other feature. This can be easily explained because if the game has no reviews, then it automatically has 0 positive reviews. When the total number of reviews is greater than 0 then the influence is less straight forward as it relies on the interplay of multiple features and the high dimensionality of the data makes it difficult to interpret this. Many leaves had a class of 5, or proportion of positive reviews between 0.8 and 1, which was the maximum, so unlike with the owner's decision tree, there is not a single branch that leads to the highest proportion of positive reviews. I would not put much value on any of these features given that their importances is just

in terms of the predictive power they have on the model, which is not a very accurate model in the first place.

Challenges and Limitations

The main challenge that I had was that extracting the data took a long time due to request limits of the APIs, and there were also some technical mishaps on my side which resulted in lost hours. This data was also complicated to clean because it had html tags, multiple values inside columns and was also multilanguage, so the cleaning also took longer than I anticipated. To get this project done in time I had to make some less than optimal decisions when I was preprocessing it, and I also did not have much time to refine the models afterwards.

One of the less optimal choices that I made was using label encoding for publishers and developers. This resulted in, for example, splits based on whether the developers' number was less than or grater than 800 which had no real meaning because those numbers were just assigned to each developer at random. A better approach would have been to use OneHot Encoding to assign a Boolean value to indicate the presence or absence of a particular developer/publisher, but there were thousands of unique values in these columns and the dimensionality would have grown even further. Also, unlike with genres, languages, categories, and tags eliminating the ones for which less than 5% had that attribute could have impacted the dataset a lot because 5% of over 80,000 rows is around 4,000 and it does not seem plausible that many, if any, developer could have made that many games. A better way to perhaps handle this would have been to create categories for developers based on size of the company, but that would have involved a lot of external research.

Another limitation of the data is that I calculated the proportion of recommendations based on total reviews, but this was not entirely accurate as some games had 0 reviews but did have recommendations. I would imagine that this is because reviews can be hidden and only available to friends, so those recommendations might have been made on hidden reviews.

Looking at the feature importances for both models, the release year of the game also seemed to play some role. There were games in the dataset that had been initially released in the mid 90s, and games that had been released just a few months ago, which might not be a fair comparison to make. Looking at games released across a smaller period of time could have led to better results because there would be less disparity when it comes to system requirements, the rising initial price of games across the years, the increasing number of steam users, the amount of time the game has been out and available to players.

Another limitation is that by using GridSearchCV to get the parameters for a pruned tree, it only iterated over the dictionary of hyperparameters set by me and though I used common numbers for these hyperparameters, there is always the possibility that there were hyperparameters that could produce more accurate trees and they were not included in the grind search.

I also used a classifier instead of a regressor for the number of ratings which involved creating arbitrary categories instead of using the actual number, and this could have impacted the accuracy of that model.

It is also worth noting that I used feature importances based on impurity, and this has a bias towards features with a high number of unique values (Sci-KitLearn, n.d-b), and that the class attribute of number of owners is in itself an estimate, so the results of these project might have inaccuracies beyond the inaccuracies of the decision tree model.

Conclusion and Future Directions

Using Decision Tree Classifiers, game success on Steam is more accurately predicted in terms of number of owners than in terms of the proportion of positive reviews.

The prediction of number of owners seems most influenced by the proportion of reviewers on Steam that recommended the game. The proportion of positive reviews, and whether was free to play, and multiplayer was also important in predicting the number of owners.

Game success on Steam in terms of proportion of positive reviews seems to be influenced by the total user reviews due to how a total number of 0 reviews automatically means it has 0 positive reviews. Beyond, this, the accuracy for this classifier is very low and close to chance levels, so I would not place much value on its feature importances.

Future directions that can be worth exploring is using feature selection to reduce dimensionality, reducing the data set to only games within a 5 or 10-year time frame, using a decision tree regressor for the proportion of positive reviews, using a single multioutput decision tree classifier instead of two independent ones, or using ensemble methods. All of these could improve the accuracy of the model and feature reduction could improve the interpretability of the tree as well.

Acknowledgements

I would like to acknowledge the blog post by Nik Davis (2019) regarding their own project using Steam and Steam Spy Data. It was through this blog that I was able to find out where to get data for the number of owners, and though I did not directly use any of their code, because this was my first-time doing data extraction in Python, I found the detailed documentation of their process most helpful in writing my own code and doing this process myself.

References

Galyonkin, S. (n.d.) *About*. Steam Spy. https://steamspy.com/about

Davis N. (2019). Gathering data from the steam store API using Python. *Nik Davis*. https://nik-davis.github.io/posts/2019/steam-data-collection/

Harris, C. R., Millman, K. J., Van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., ... Oliphant, T. E. (2020). Array programming with NumPy. *Nature, 585*, 357–362. https://doi.org/10.1038/s41586-020-2649-2

Hunter, J.D. (2007). Matplotlib: A 2D graphics environment, *Computing in Science & Engineering, (9)*3, 90-95. https://doi.org/10.5281/zenodo.10916799

McKinney, W. (2010). Data structures for statistical computing in Python. *Proceedings of the 9th Python in Science Conference (445)*.

Pandas Development Team. (2024). Pandas-dev/pandas: Pandas (v2.2.1). Zenodo. https://doi.org/10.5281/zenodo.10697587

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M. Prettenhoffer P., Weiss R., Dubourg, V., Vanderplas, J., Passos, A. Cournapeau D., Brucher, M., Perrot M, & Duchesnay, É. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research, (12)*85. https://jmlr.csail.mit.edu/papers/v12/pedregosa11a.html

Richardson, L. (2015). Beautiful soup documentation. *Beautiful Soup*. https://beautiful-soup.readthedocs.io/en/latest/

Sayer, M. (2022). *The 19-year evolution of Steam*. PC Gamer. https://apastyle.apa.org/style-grammar-guidelines/references/examples/webpage-website-references

Sci-Kit Learn. (n.d.-a). *Decision Trees*. https://scikit-learn.org/stable/modules/tree.html#tree

Sci-Kit Learn. (n.d.-b). *Decision Tree Classifier. https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html*

Appendix A: Summary Table of Data Cleaning Process

Original Data	Clean Data
SteamAppID (int)	SteamAppID (int)
Type	[Eliminated]
Name (string)	Has name (bool)
Required age (string)	Required age (int)
Is free (bool)	Is free(bool)
Detailed description (string)	Has detailed description (bool)
About the game (string)	Has about game (bool)
Short description (string)	Has short description (bool)
Supported languages (list)	Individual language columns (bool)
Header image (string)	Has header image (bool)
Capsule image (string)	Has capsule image (bool)
Capsule image v5 (string)	Has capsule image v5 (bool)
Website (string)	Has website (bool)
Pc requirements (dictionary)	Processor requirements, Memory requirements, Graphics requirements, Storage requirements (categorical)
Mac requirements (dictionary)	
Linux requirements (dictionary)	
Developers (list)	Developers (categorical)
Publishers (list)	Publishers (categorical)
Packages (list)	In package (bool)
Package groups (dictionary)	[Eliminated]
Platforms (dictionary)	Linux, Windows, Mac (bool)
Metacritic (dictionary)	Has Metacritic score (bool)
Categories (list)	Individual categories columns (bool)
Genres (list)	Individual genre columns (bool)
Screenshots (dictionary)	Has screenshots (bool)
Recommendations (dictionary)	Total recommandations (int), proportion recommended (float)
Release date (dictionary)	Coming Soon (bool), Release date (datetime)

Support info (dictionary)	Has support info (bool)
Background (string)	Has background (bool)
Background raw (string)	Has background raw (bool)
Content descriptors (dictionary)	Has content descriptor (bool)
Ratings (dictionary)	[Eliminated]
Controller support (string)	[Eliminated]
DLC (list)	Has DLC (bool)
Demos (list)	Has demos (bool)
Movies (list)	Has movies (bool)
Achievements (dictionary)	Total Achievements (int)
Reviews (string)	Has professional reviews (bool)
Legal information (string)	Has legal information (bool)
DRM notice (string)	Has DRM notice (bool)
External user account notice (string)	Has external user account notice (bool)
Positive reviews (int)	Proportion positive (float), Proportion negative (float), Total reviews (int)
Negative reviews (int)	
Number of owners (string)	Number of owners (categorical)
Average playtime in minutes (int)	Average playtime in hours (float)
Median playtime in minutes (int)	Median playtime in hours (float)
Initial price in cents (int)	Initial price in USD (float)
Tags (dictionary)	Individual columns for each tag (bool)