An Analysis of Factors that Contribute to a Successful Steam Game

Matthew Dinh
madi0567
CSCI 4502
Matthew.Dinh@colorado.edu

Samuel Mauck
sama3973
CSCI 5502
Samuel.Mauck@colorado.edu

Chase Dudas
chdu9122
CSCI 4502
Chase.Dudas@colorado.edu

Connor Cook
coco0808
CSCI 5502 (Distance)
Connor.Cook@colorado.edu

Figure 1: Steam Logo

ABSTRACT

Steam is the largest digital distribution service platform for PC video games. With over 30,000 unique titles, it is the central platform for PC games. This project seeks to analyze Steam's catalog and determine which features of a game make it successful. While success itself is hard to define, this project examines a variety of factors and their impacts on statistics traditionally correlated with success such as high ratings. Previous work used high viewership as its measure of success, which biased it towards games that were popular, leaving out smaller games that could also be considered successful. The end result of this project is a regressor that can be used to predict game success which we define by player reviews.

ACM Reference Format:
Matthew Dinh, Samuel Mauck, Chase Dudas, and Connor Cook. 2019. An Analysis of Factors that Contribute to a Successful Steam Game. In *Proceedings of CSCI 4502/5502*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

CSCI 4502/5502, December 12, 2019, Boulder, CO, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Video-games have risen in popularity over the years, especially due to the recent growth of the eSports industry. However, one question has haunted game developers since the first commercial games have been developed: is their game successful? While success is highly subjective, there is no doubt that success has changed its meaning throughout the years. Perhaps previously, it was highly based on critic scores, whereas now some may consider viewer count to be a more important metric.

The first major contribution of this work will be the collection and cleaning of a large data set containing a wide variety of information on video games. Through the analysis of the collected data, we hope to provide an objective regressor that can be used to identify the success of current games and to predict the success of future games before launch, which will be the second contribution of this work. Analyzing how the regressor uses the different aspects of a game will determine which features are most important to the success of a game, and this analysis will be the third contribution of this work.

1.1 Motivation

Many video game enthusiasts and critics like to predict the success of a soon-to-be related video game. While success is highly subjective, there are many games which are classically regarded as "successful" in the industry.

It would be intriguing to discover which factors contribute to the success of these games, and if these factors have shifted over

time. Furthermore, it would be interesting to discover if traditional algorithmic or classification techniques would be able to accurately categorize a game as successful or not using the vast amount of data on Steam.

In terms of real world application, such an algorithm could help predict whether or not a game will have a successful launch. This is valuable for shareholders and gamers alike, who could use such an algorithm to make in informed decision as to whether one should invest in the company that makes the game, or to decide whether or not you want to pre-order the game.

The model or algorithm would also be useful to developers, as it would help them know how to alter their game to be more successful and help them know which kind of game to start on next. It could also be a valuable tool to the teams that market games, because it would help them know which aspects of a game to emphasize to get a better response from their advertising campaigns.

1.2 Challenges

There are a couple challenges for this specific project. First and foremost is the question of causality. This project aims to use features about a game in order to build a regressor to predict success. However, the casual direction is important. For example, is a game successful because it has good reviews, or does the game have good reviews because it is already successful? The team will need to carefully consider this when constructing the regressor.

Another potential challenge is the data itself. Unlike other works which focus more on player behavior data, this project aims to the use the raw features about the games themselves in order to predict success. This could include features anywhere from tags to game specific metadata. There is the potential for a lot of the features to have no correlation with the success metric, so finding a diverse, representative set of features could potentially prove challenging.

One final challenge is the data collection itself. This project plans to scrape game data from a third party site that does not provide an open API, so most of the scraping logic will need to be written and performed from scratch.

2 RELATED WORK

While there has been prior work evaluating Steam data sets, most of it focused on player behavior rather than on the games themselves. Perhaps the largest analysis of Steam data to date was performed by O'Neill et al. [6]. Their team analyzed the entirety of the Steam player-base at the time of their analysis, looking at social structure, game ownership, and game playtime. Similarly, Sifa et. al [7] analyze "the relationship between game ownership, time invested in playing games, and the players themselves." However, little formal work has been done analyzing the games themselves, save for the work of Michal Trněný.

2.1 Michal Trněný

Michal Trněný, a researcher at the Faculty of Informatics at Masaryk University, Brno presented his master's thesis which employed very similar methods and analyses as to what this project aims to achieve [8]. Like this project, he self-gathered a data set using a combination of the official Steam API and third party sites. He utilized machine learning techniques to develop a variety of regressors such as random forests or linear regressors in order to predict game success. He used a combination of over 200 features to develop these regressors. As such, his work will serve as an invaluable reference during the development of the team's project.

There are a few key differences between Trněný's work and this proposed project. First, this project aims to gather a larger amount of data by scraping the majority of SteamDB. This site offers much more information than the official Steam API that provided the bulk of the data set Trněný used in his work. Secondly, Trněný used average playtime as his main measure of success. This project plans to use review data as the main success metric. The extra data gathered, plus the different evaluation metric, should allow this project to be a meaningful and stand-alone extension of the work that Trněný initially laid out.

3 METHODOLOGY

The goal of this project is to create a regressor that can be used to predict the success of a game from data on Steam. Success in this case is measured by the average review score of the game. The work being proposed is to create a data set from official Steam data and other sources, ensure the data set is suitable for building regressors with, test multiple regressors on the problem, and finally to determine which of the tested regressors is best suited for solving this problem.

The team is among the larger in the class, with two undergraduate students and two graduate students. The scope of the work is also fairly large, but there are pieces of the work (such as evaluating games based on news posts and player review text) that can be added to this work if there is extra time, but are not necessary for the project to produce a result. Given these facts, this work is both sufficient and feasible for a group of our size and experience.

3.1 Data

The team is going to scrape data from Steam using the Steam API[2] and from third-party sources such as SteamDB[9] to create the dataset. Some variables that the team plans to look at are type of game, concurrent players, game developer, ratio of positive to negative reviews, and many more. The data will be cleaned to handle values that are empty or otherwise unusable.

3.2 Subtasks

The proposed work can be broken down into the following subtasks: scraping data, exploring the data, cleaning the data, training regressors, and comparing regressor performance. The subtasks are described in more detail in the following paragraphs.

3.2.1 Data Scraping. The team plans to scrape data from the Steam service itself, via the Steam API[2], and from the third-party website SteamDB[9]. An example page of a SteamDB page is shown in figure 2

The official Steam API will be used in order to obtain a full catalog listing of all game currently on the store, and SteamDB will be used to provide a wide variety of data on the games themselves, including (but not limited to) player count over time, game price over time, average playtime for a player, player reviews, developer name,

Figure 2: An example game listed on the third-party website SteamDB

and publisher name. SteamDB will be scraped using custom software employing, primarily, the Python web modules requests[5], cfscrape[1], and pyquery[3].

3.2.2 Data Exploration. After scraping the data, the team plans to spend some time examining it. This will including looking at the distributions of the data, minimum and maximum values, and correlations between variables. This stage will also include deciding which variables are useful in determining whether a game should be considered successful or not. We expect that ratio of positive to negative reviews will be our primary measure of success.

3.2.3 Data Cleaning. During the team's exploration of the data, the team will find inconsistencies or other problems with the data that need to be altered or fixed for the data to be usable. This subtask will involve task such as handling missing values, normalizing the data, and combining the data sets we get from different sources into a single useful data set. Given extra time, it may also include building language models on any text data we collect, to make it usable as an input to regressors.

3.2.4 Regressor Training. By this point the team should have a clean, usable data set. Before the actual training, the team will need to decide which metrics to use in the regressors. The team will look at metrics such as mean absolute error, mean squared error, area under ROC curve, F1 score, and others that the team finds. This task will then consist of creating different regressors, training them on the data set, and collecting the results from the testing and validation data sets. Some regressors we will try include linear regression, ridge regression and SVM adapted for regression.

We believe that the important variables that will contribute to predicting a successful game include user reviews, game genre, and game developer. For games with a larger publisher some other factors are important, such as player peak and average viewer count.

3.2.5 Regressor Evaluation. The last subtask in the project is evaluating the different trained regressors to determine which are useful in predicting Steam game success. This will be done by comparing the regressors with the different metrics we identified in the Regressor Training stage.

4 METHODOLOGY

4.1 Dataset Construction

The dataset was constructed in three phases. First, the official Steam API was queried in order to a obtain a list of all game and DLC app IDs available on the Steam store. Second, these IDs were used to scrape web data from SteamDB. Finally, the data obtained from SteamDB was combined with supplemental data obtained from the official Steam API.

4.1.1 App ID Gathering. The dataset was compiled by scraping the third party website SteamDB, and by making API calls to the official Steam API. First, however, the official Steam API was queried to return the app ID for every game and DLC on the store. The API endpoint:

https://api.steampowered.com/IStoreService/GetAppList/v1/

Returns a JSON array keyed on `apps`. The API endpoint requires the following parameters as a part of the query string payload: `key`, which was set to our Steam API key; `include_games`, which was set to `1`; `include_dlc`, which was set to `1`; and `max_results`, which was set to the max allowable value by the Steam API, which is `50000`. The results were paginated by looking for the presence of the `have_more_results` variable in the response, and using the included `last_appid` in the payload for all subsequent requests. At the time of writing, there were a total of 59129 app IDs (which includes both games and DLCs) returned by the API.

4.1.2 Web Scraping. The main crux of the dataset construction relied on web scraping, as SteamDB did not, and does not at the time of writing, offer any sort of official API to query their data. A specific combination of Python modules made this less of a daunting task. First, the Python `requests` module provides an interface to retrieve HTML content from remote websites. In addition, it provides built in session management to ensure that any needed cookies or headers are set and sent correctly. The `cfscrape` provides an interface that extends the `requests.Session`. This module hooks into a local instance of NodeJS to provide the session class with additional functionality, allowing it to detect and respond to CloudFlare challenges that would otherwise stop the scraper in its tracks. Finally, the HTML content returned from the `requests` module was passed to the `pyquery` module, which provides a subset of jQuery-like selectors, allowing for easy retrieval of content inside the webpage.

For every ID gathered in the previous step, the HTML for the webpage:

https://steamdb.info/app/###/

was returned, where ### refers to the current app ID at scrape time. From there, the scraper pulled out individual sections from the web page, and for each "feature" on the webpage, ran it through a custom built value transformer. This value transformer served as an early prepossessing mechanism, ensuring that the dataset was at least somewhat readable. For "features" (which on the website usually corresponded to entries in a table) that were common

amongst multiple games, the name of that feature/row header was explicitly defined inside the transformer class. This transformer class mapped between feature names and higher order functions that would be performed on said feature at scrape time. For example, the `metacritic_score` feature was defined inside the `int_attributes` feature list. Thus, for example, whenever an attribute was pulled out of the HTML and appeared in that list, the higher order function `__val_to_int` was applied on it, converting the text to an integer to be stored in the actual dataset. Certain features were not explicitly defined, but if they matched a RegEx pattern, for example, had a function applied to them anyway. If a feature didn't match either (since its impossible to pre-define all of the features, as they vary from game to game), plain text was simply returned.

4.1.3 Official Steam API. At scrape time, after a JSON document was generated based on the webpage HTML, the official Steam API was queried to provide supplemental information. The API endpoint:

https://store.steampowered.com/api/appdetails?appids=###

returns some information about the game, where once again ### refers to the App ID at scrape time. Extra measures were taken to ensure that the official Steam API did not overwrite data from SteamDB. Some of the data it provided was redundtnat, such as the `screenshots` feature, for example, but for the sake of having a complete dataset the team opted to keep both.

4.1.4 Dataset Concatenation. As noted previously, for each game, a JSON object was generated. As such, for the team's choice of "database", we simply used a JSON array, where each entry in the array was a game. However, this choice was problematic, as in order for an element to appended to a JSON array, the JSON must fully be parsed, which entails loading the entire dataset into memory every time a new game is added. With the final dataset coming out to about 11GB, this was simply not an option. Instead, as a trick, instead of loading the entire JSON file, in order to write a new entry to the "database", the code did the following operations: Seeked to the end of the file, went back one more character (so the "cursor" was on the closing] of the entire array), and appended the new entry, and re-added the closing]. This ensure that the data did not have to be read into memory every time a new game was added, while still maintaining the validity of the JSON format.

However, a similar issue arose when attempting to load the dataset into Pandas for analysis. In order to load the data on a chunk by chunk basis (since, once again, the entire dataset is too big to load into memory altogether), Pandas expected each JSON object to be on its own line, a format called JSONL. However, the dataset produced by the scraper was all on one line. As such, a script was written that chunked the data and inserted newlines in the appropriate places to convert the data to JSONL.

4.1.5 Main Code. In order to avoid any sort of rate limiting, and to combat any intermittent network issues, the main part of the scraping algorithm used exponential backoff, up to a maximum of 15 tries, when making the outgoing web requests. This ensured that one error would not crash the entire scraper, and it ensured that there was a better chance of getting the data. In addition, the code made note of every app ID that failed more than 15 times. In total, 39 IDs failed, leaving the team with a final total of 59090 entries in the dataset.

4.2 Dataset Preprocessing

In the data set, there is about 1.3 gigabytes of data and to process all of it manually would take a very long time. Therefore, the preprocessed data set had categories removed and many categories transformed. Eventually, all of the data being passed into the classifier needed to be converted into an integer format. As we examined the data, we found five ways that we wanted to interpret the different features. They were: feature presence, feature count, feature value, categorical features, and time-series feature summary. These examinations resulted in the creation of a new sub-dataset comprised of the basic features that we will use to evaluate a game's success. In addition to these basic features, more complex advanced features were derived from combining, manipulating, or evaluating two or more of the basic features. As we advance, feature elimination of the sub-dataset will also become important. Checking for feature-to-feature and feature-to-label relationships, especially correlation, will give us insight about whether or not our basic/advanced features are actually useful or give us any knowledge. Using different models and APIs we can make these comparisons easily and get a lot of information on each data point.

4.3 Feature Selection

To begin, we selected categories based on our own domain knowledge that we empirically felt were good features that could affect the success of a game. Then, we removed categories that we felt had no contribution to the game's success. There were also some features that we were unsure of what it represented. There were some important thoughts when it came to selecting these features. One of which was whether or not the feature was important only for AAA games where there were more likely to be more of that feature. For example, player count would be drastically different for a AAA game compared to a small indie game. Therefore, it would be difficult to use such features to classify games of different sizes.

4.3.1 Feature Presence. In the data set, there were some features where we were solely concerned with whether or not it was present in the game data. So, in the data set, a feature is considered present if there a value of 1 and 0 for a feature that is not present.In addition, a game that has a nan value in the data set will be defaulted to 0. Consider a person who enjoys to play video games at the comfort of their couch. A game that has controller support may do better than a game that does not have that feature.

4.3.2 Feature Count. Feature count represents the count of feature. For example, a game may have a list of achievements, and so feature count would represent the amount of achievements there are for that game. In addition, games that have NaN values are defaulted to a count of 0. The data is then normalized so that the numbers do not get too big. This is an interesting feature such that a feature that has a higher count may have a stronger impact on the success

of a game. This feature is especially prone to games that are backed by a larger company.

4.3.3 Value Features. Value features are descriptive features that when casted to numerical values become important to our data set. These features appear in our database as strings, booleans, floats, and dictionaries. For each occurrence, it was crucial that we interpret this data a numerical values for our evaluation to work effectively.

One example of important value features are items that were stored as a string but contain essential numeric information. For example, AvgPlaytimeTotal is scrapped from SteamDB as a string in the format of a numeric followed by a string that is either hours, minutes, or seconds. AvgPlaytimeTotal is a meaningful feature because it shows player retention for a given game which can make a game more successful. Because this information is stored as a string, it makes it really hard to evaluate based on this feature. After processing AvgPlaytimeTotal as a value feature, it gets stored as an value of the total number of average hours a player spends on a game. This makes AvgPlaytimeTotal uniform across all games and allows us to effectively use it to evaluate success.

Another example of value features contained in our data set are True/Yes False/No values. Take for example the ContainsAdultContent feature. In this feature the values are stored either as Yes or as No. Although this is easily interpretable by us, our analysis on the data set would have a hard time evaluating this item. Instead, we took this string and converted it to a boolean to easily store a Yes as True and No as false.

The last troublesome set of features in our data set involve ranges of numeric values stored as strings. Take for example the feature owners, which is stored from SteamDB as a range of potential owners separated by ellipses. This value directly correlates to a games success so it was important to process this feature into something usable. From the raw input, the values were averaged and stored as one numerical value to represent the average estimated owners of a game.

Value features are important to our data set because they turn interesting features into usable ones. Ultimately, these values focused on converting values that are easily understandable in our eyes into something usable to run our evaluations on.

4.3.4 Categorical Features. Categorical features are features where one feature consists of one or more categories. We chose to represent these features using one-hot vectors. A one-hot vector is where a single feature with categorical values is turned into a vector with one column for each categorical value. For example, a feature that had values 'Red', 'Blue', and 'Green' would be turned into a vector with three columns, where a 1 in the first column means the row's value was 'Red', a 1 in the second column means the row's value was 'Blue', and a 1 in the third column means the row's value was 'Green'. A 0 in one of the columns means the row does not have the corresponding value.

Some models want a single column dropped to prevent correlation among the attributes for most one-hot vectors, letting a 0 in every other column of the vector indicate that the row had a 1 in the dropped column. In our case it is possible for a single row to be positive in more than one category for most of the categorical features, so we did not drop any of the columns. Some games did not have any values for certain categorical features, so they were assumed to have a 0 in each of the columns of the one-hot vectors resulting from those features.

For the 'Developer' and 'Publisher' features, there were too many values, roughly 25,000 publishers and 20,000 developers. A one-hot vector of these values would have overwhelmed the other features, so we pruned them down to developers that created at least 25 games and publishers that had published at least 50 games. This left us with 35 developers and 44 publishers.

Another categorical feature with a potentially troublesome number of values is 'Tags', which has 378 values. We haven't trimmed this down, but will in a manner similar to how 'Publisher' and 'Developer' were trimmed down if it turns out to be a problem.

4.3.5 Time-series Feature Summary. Time-series features are features such as price history over time or number of players over time. Intuitively, these features are very important, especially in a temporal context. The repeated presence of a game being on sale, for example, could potentially influence the rating. However, as noted previously, the input to a regressor can only be numeric data. The more common regressors have no notion of time. As such, to approximate the impact of these features, summary statistics were applied to all of the time-series data. Specifically, the min, max, mean, and median were gathered for the price history over time, the number of players over time, and the amount of Twitch viewership over time. Additional care was taken to avoid any NaN values, and to appropriate defaults (0, in this case), in the event that these particular features did not exist (a free game wouldn't have price history, for example).

4.4 Regressor Training

We experimented with many of the regression models that are available in Scikit-Learn. These include linear regression and an SVM adapted for regression.

We also experimented with different error types when training, including ones such as absolute squared error and mean squared error.

We set aside 10% of the data to use as our test set and trained on the other 90%, using 10% of the training set as the validation set, which let us know when to stop training.

5 EVALUATION

For our evaluation metric, we measured how successful a game is based on the ratio of upvotes/positive reviews a game receives to downvotes/negative reviews on Steam. Other metrics such as revenue, Twitch viewers, or player retention are less likely to show success because these metrics bias our results towards games made by more developed businesses. Potential future work is creating a regressor specifically for these more developed businesses, which could then use metrics such as revenue and Twitch viewers to show success.

Our evaluation ranked the most up-voted, positively reviewed games towards the top and the most down-voted, negatively reviewed games towards the bottom. One metric we used to compare our regressors is the correlation coefficient, often known as r-squared. Ideally we would have seen a value near 1, which would indicate to us that our model was able to predict very precisely the

ratio of positive to negative reviews. In other words, this would tell us that our regressor was able to accurately predict the level of success of different games. Unfortunately we did not see a value anywhere near 1 in our work.

A potential baseline to compare our regressors against involves evaluating our results against the results of randomly ranking the same games, which has an r-squared value of 0.0. Another potential baseline would be assigning each game the average of all games, or all games within a subsection of the data, and ensuring our model gets closer to the real value than the average does. This did not get much better than the random baseline, indicating that the problem we are trying to solve is a difficult one. Ideally we would be comparing against the model created by Michal Trněný[8], but that does not appear to have been uploaded somewhere we can access it.

5.1 Experimental Setup

Given the variable and unknown nature of the problem, the team tried various experimental setups when training the regressors, each iterating on the last in one way or another. These setups are described below.

5.1.1 PCA. The first setup the team tried was to pre-process the dataset using a simple Principal Components Analysis (PCA). We chose 95% as the amount of variance that the principal components needed to explain, and this resulted in 9 features being generated from the 400+ we started with. From there, this data was passed to a linear regressor and a support vector machine regressor.

5.1.2 Variance Thresholding. The next experimental setup was another form of feature reduction, this time using scikit-learn's `VarianceThreshold` transformer, which removes features whose variance are under a certain threshold. Features that do have a low variance do not contribute much to our regressor model so we are able to throw these features away. In our case, we used a threshold of 80% such that any features that do not vary more than 20% will be thrown away. This was then passed to a linear regressor.

5.1.3 L1 Regularization. The next experimental setup was setup using scikit-learn's `SelectFromModel` using LassoCV as the base estimator. scikit-learn's's `SelectFromModel` is a meta-transformer that selects features based on importance weights. The method takes in an estimator which the transformer is built from and a threshold value which will check if a given feature's importance is greater than the thresholds. LassoCV is a linear model with iterative fitting along a regularization path. The code will iterate through the features and use LassoCV on the feature's data to check if it reaches the threshold value. If the feature matches the criteria, the current feature count will increment and continue on until the amount of features reaches a user specified amount of features or until all the features have been processed through. The setup for our model uses a threshold value of .25 and a user-specified feature count of 465.

5.1.4 RFECV. The next experimental setup took a sharp deviation from that of the previous setups. While feature elimination was still the main goal, this setup went about it in a different way. First, the data was pre-processed using scikit-learn's `MaxAbsScaler`. This transformer scaled each column such that the maximum value of that column would be 1. The important part about this pre-processing technique is that it is designed to preserve sparsity by not shifting the center of the data. A large part of our overall training set is spare data (because of the one-hot vectors), so this needed to be preserved while also ensuring that the numeric features did not overwhelm them. Next, RFECV, or, recursive feature elimination with cross validation, was performed using scikit-learn's `RFECV`. This class starts with all of the features, trains the model (which in the case of this experimental setup was an SVM regressor with a linear kernel) and performs 5-fold cross-validation. Then, one feature is dropped, and this feature is repeated until there is only one feature left. Finally, the optimal number of features (where the cross-validation score is highest), is reported. From there, the model can be re-trained on the full training set (where the validation set is not withheld), but this time on the subset of optimal features discovered. The results of the RFECV for this experimental setup is shown above in figure 3.

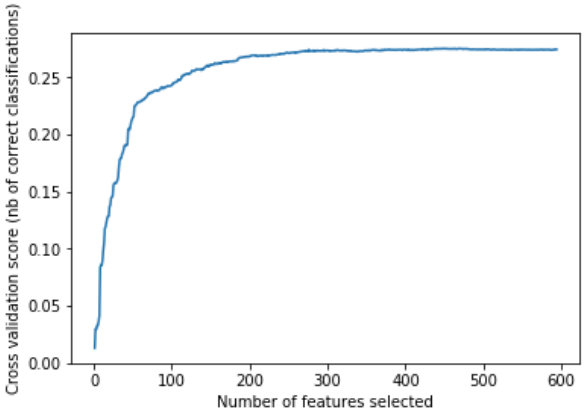

Figure 3: Feature Elimination. This graph shows the cross-validation score of a linear SVR as the number of features selected from the dataset increased. The score was very near its peak at about 150 features selected, implying that many of the features are not important in predicting a game's success. The actual peak score was at 448 features.

5.1.5 Ensemble Regressors. The next experimental setup was a set of ensemble regressors. Up until this point, the scores on the previous setups had been constantly very lower, hovering around the 0.1 - 0.2 range. As such, the team opted to try a set of ensemble regressors. Three regressors were each trained on a subet of domain specific data, rather than trying to have one regressor deal with the wide range of domains present in our training set. The first regressor was a Random Forest (which, on its own, is an ensemble of itself), and this was trained on the set of one-hot features (the tags, categories, genres, etc.) Next, a Gradient Boosting Regressor was trained on the set of time series features. Finally, another Gradient Boosting Regressor was trained on the set of boolean features. After each model was trained on the smaller subset of data, a search over each model's hyper-parameter space was performed using scikikt-learn's `GridSearchCV`. By using the validation set as a proxy for

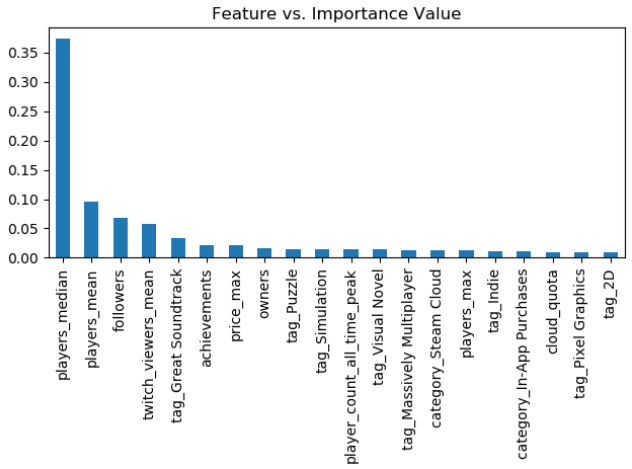

Figure 4: Feature Importance of the top 20 features extracted from the fitted GradientBoostingRegressor model.

Model score on test set

Model	Score (R-squared)
Linear Regressor w/ PCA	0.28
Support Vector Machine w/ PCA	0.09
Support Vector Machine w/ RFE and MAS	0.28
Voting Ensemble	0.38
Gradient Boosting Regressor	0.47

Table 1: This table shows the results of different models. It also includes references to the data pre-processing methods used in training. The model that performed the best was the gradient boosting regressor with no data pre-processing. The different pre-processing methods are as follows: PCA stands for Principal Component Analysis, RFE stands for Recursive Feature Elimination, and MAS stands for Maximum Absolute Scaler.

the test set, we were able to tune the hyper-parameters of each individual regressor. Finally, these three models were aggregated into scikit-learn's `VotingRegressor`, which averages the result of each individual model's predictions. Additional weights were passed into the `VotingRegressor` model to give more weight to the individual models that were doing better on the validation set.

5.1.6 GradientBoostingRegressor. The final experimental setup was just the GradientBoostingRegressor alone. This was used as a part of the ensemble model because it had good performance on the subsets it was trained on, but the team had yet to try it on the full dataset as a standalone classifier. Specifically, two forms of gradient boosting regressors were tried. The base scikit-learn `GradientBoostingRegressor` and the experimental `HistGradientBoostingRegressor`. The former is based off of Microsoft's LightDM implementation and works better on larger datasets. During testing on the validation set, the `HistGradientBoostingRegressor` did perform slightly better, however its memory usage was much higher, making it much harder to perform grid search to optimize the hyper parameters. The distinction, which led the team to choose the `GradientBoostingRegressor` instead, is that it ranks the features in terms of their importance, something that the histogram variant does not do. This turned out to be the best performing model, and as such the team extracted the most important features, which are shown in figure 4

5.2 Results

Running the features generated via PCA through a linear regression model resulted in a, R-squared value of 0.28 on our validation set. While low, this result showed us that the features we had in our data set did have some predictive power. We tried running these same features through an SVM regression model (using various kernels), but the predictive r-squared in these cases ended up being about zero, which indicated that either the model itself wasn't well-suited to the task or that we were unable to tune the model well.

Once we removed the features that had low variance, it increased the linear regression score but by very little. It increased it by .01-.03 but due to the variation already with the linear regression score and the little increase, it was hard to attribute the score increase to the low variance removal. This may show that many of the features had little variation and so many of the features remained not affecting the linear regression score by much.

The results of combining SelectFromModel and LassoCV was practically inconclusive as the variance of the linear regression model was wide. The linear regression score varied from -2 to +0.32 and was very inconsistent at hitting the +.32 range. We are unsure why the linear regression score varied so much with the feature elimination method, but we would say that given more time at configuring the different options of LassoCV, there may be a way to get a consistent linear regression score from it.

At this point we decided to try another model, so we moved on to a linear Support Vector Machine regressor (or SVR). Our first SVR performed quite poorly, only getting an r-squared value of 0.09. This was with the PCA-generated features that were originally used with the linear regressor. This led us to change the pre-processing we used in an attempt to improve the score. The best pre-processing for the SVR was to first run a Maximum Absolute Scaler on the data (which is a way of normalizing sparse data) and then to run Recursive Feature Elimination. This second pre-processing method works by dropping each feature, running five-fold cross-validation on the model, and then at the end returning the features that resulted in the best score. This pre-processing brought the SVR's score up to 0.28, which was improvement over the original SVR, but just managed to match the original linear regressor.

Since the simple models were not producing good results, we tried multiple other models and model ensembles. One attempt that produced decent results was a voting ensemble that consisted of a random forest regressor trained on the data set's categorical values, a gradient boosting regressor trained on the data set's summarized time-series values, and another gradient boosting regressor trained on the values not covered by the first two models. This led to an r-squared value of 0.38, a significant improvement over our previous best of 0.28.

Because the gradient boosting regressor (GBR) played such an integral part in the voting ensemble, we decided to try training a single GBR on all the data. This led to the best model we found,

which had an r-squared value of 0.47. Strangely enough, the best value we found was with no data normalization or pre-processing. The GBR is a series of decision trees, so perhaps it needed to have a large number of features to be able to generate more decision trees to provide more accurate decisions.

6 DISCUSSION

6.1 Lessons Learned

A lesson learned came from the difficulty of dataset preprocessing. There was lots of problems that arised like some of the data came out as strings, some data passing through when it was not suppose to, and making sure there were not any strange values in general. It was nice to have other team members as we were able to split some of the dataset preprocessing techniques between the members heavily decreasing the amount of time overall. However, through this difficulty, we learned the importance of having a good dataset before putting the data through classifiers. Having a strange value can create errors, and cause inconsistency in the model.

One lesson learned is that tuning a model is more of an art than a science, and that it can take quite some time to get a specific model working well. There are some models that we were not able to get working well at all, but it seems more likely that the problem lies in the hyper-parameters we set for the models rather than the models themselves. Conversely, it is also possible to spend far too long tuning hyper-parameters hoping for a better outcome from a model that simply is not going to get good results. Knowing when to keep going and when to stop seems like something that needs to be intuited from experience.

Another lesson learned is that, in data mining projects, it is less likely that one will be able to drastically out-perform the work that has come before. It is far more likely that, after lots of hard work and a few flashes of insight, you will achieve only modest gains. This is generally more true when working on heavily studied problems, but even when there is only a small body of prior work making significant strides is both difficult and requires some luck.

An additional lesson is just the sheer amount of time it takes from gathering the data to finally creating a model for the data. To begin, it took a while to create the dataset originally as there was lots of data on SteamDB to compile together. It took around 30-40 hours to compile the data together and due to complications, we had recompile the data twice which ate up a lot of time before we could even preprocess the data. In the interest of time, we decided not to include DLC data either as that would have increased our dataset size immensely and increase the time taken as well. Then, doing the preprocessing techniques on the dataset took about 4-5 minutes to create the dataset that we would put through our classifiers. Then, depending on the classifier, that could take 3+ minutes per iteration. Overall, we learned that data mining is a time consuming process. Some things that we could have done differently to compensate for this is to simply start earlier, and devote more time to the project in general.

6.2 Limitations

One limitation is that the data set was only composed of data directly from the Steam API[2] and from SteamDB[9]. There are doubtless other sources we could find that contain unique features useful to our task. An example source is SteamSpy[4], which was used by Michal Trněný[8].

Another limitation is the time we had to spend on the project. If we had longer to examine the data and experiment on it, we would have been able to obtain deeper insights into which factors are meaningful in determining the success of a Steam game and which factors are not. A semester of time split between different classes and work really isn't all that much.

In the end, the features that showed to be important in the success of a game ended up being features that indicate popularity. For example, the top three features that made up our model were the median number of players, mean number of players, and number of followers. This is something that we wanted to avoid because it would not accurately model smaller games' successes. It would be interesting to create a model that removed the features that indicated popularity. Based off the graph referenced before, the new top three features would be great soundtrack, achievements, and price max which is much more relevant to games in general rather than just popular games.

6.3 Future Work

There are a number of other tasks that could be completed to further better our results. Although we were able to reach a conclusion, given more time there are multiple processes we could have used/trained other techniques to better process and analyze our data. Processes like text analytics, discretization, better prepossessing, and including more features could be useful for adding additional data points to our set.

One process to be done in future work is text analytics. This process can be used to draw meaning out of written communication. Text analytics uses AI with NLP (Natural Language Processing) to transform free text into normalized data suitable for analysis. We can use text analytics to process text from developer news posts and customer game reviews. Text is first preprocessed using tokenization, lemmatization, stemming, and the removal of stop words. This produces a set of text that contains only important words. A language model can then be built using this set of words. This language model can then be clustered with other data to help determine the worth of a game. Further text analysis can occur via techniques such as topic modeling.

Another process to be done in the future is to include more features in our analysis. This would give us more data points and could reveal come hidden correlations. We shrunk the number of included items to include on the main relationships we deemed important. That being said, we are sure that there are countless other relationships that could be meaningful in the context of rating a game. On similar terms, in the future we could also preprocess the data better. Our analysis could be over looking connections simply because our data was not as well formatted as it should be.

Going off of feedback from prior work, we could also expand our data in the future by pulling from sources like game DLC's or other websites. This would give us more data points to analyze and combine with our current data from Steam. This would require us to do more preprocessing and adjust our algorithms, but we believe this would be worth the extra effort.

6.4 Applications

The applications of this classifier is that it can help developers predict the success of their game. For example, if a developer is creating a game, then through this classifier, they would be able to see what features they are missing which may impact the success of their game. So, the developer will implement this missing feature and theoretically receive better success.

Another point of view is through the lens of a customer who is looking to purchase a game. One of the problems with preordering a game is that the customer is unsure a game will actually be good so they may be discouraged of preordering the game. However, with an accurate classifier, the customer can run the upcoming game through the classifier and be relatively certain on the success of the game.

7 CONCLUSION

The team experimented with multiple regressors and data preprocessing techniques in order to find a combination that is best able to predict the success of a Steam game (where success in this context refers to review scores). The best regressor makes use of both static features that are present at the time of a game's launch (such as its genres, tags, developers, etc.), which is useful for predicting success before or right at launch, as well as features that change during the game's market cycle, such as playtime, price history, and ownership. This makes the model a useful tool for publishers, developers, and users alike, especially given how hard of a problem this has been in the past. While the model does explain around 50% of the variance in the label, there is still lots of work to be done on this regression problem, and this paper hopes to serve as a basis for future models to come.

REFERENCES

[1] et al Anorov. 2019. Retrieved October 10, 2019 from https://github.com/Anorov/cloudflare-scrape
[2] Valve Corporation. 2019. Retrieved October 9, 2019 from https://steamcommunity.com/dev
[3] et al Gael Pasgrimaud. 2019. Retrieved October 10, 2019 from https://pyquery.readthedocs.io/en/latest/
[4] Sergey Galyonkin. 2019. Retrieved December 10, 2019 from https://steamspy.com/
[5] Nate Prewitt Kenneth Reitz, Cory Benfield and Ian Cordasco. 2019. Retrieved October 10, 2019 from https://requests.readthedocs.io/en/master/
[6] Mark O'Neill, Elham Vaziripour, Justin Wu, and Daniel Zappala. 2016. Condensing Steam. *Proceedings of the 2016 ACM on Internet Measurement Conference - IMC 16* (Nov 2016). https://doi.org/10.1145/2987443.2987489
[7] Rafet Sifa, Anders Drachen, and Christian Bauckhage. 2015. Large-Scale Cross-Game Player Behavior Analysis on Steam. https://www.aaai.org/ocs/index.php/AIIDE/AIIDE15/paper/view/11542/11379
[8] Michal Trněný. 2017. Machine Learning for Predicting Success of Video Games. (2017). https://is.muni.cz/th/k2c5b/diploma_thesis_trneny.pdf
[9] xPaw and Marlamin. 2019. Retrieved October 9, 2019 from https://steamdb.info/

A HONOR CODE PLEDGE

On my honor, as a University of Colorado Boulder student, I have neither given nor received unauthorized assistance.

B WORK DONE BY GROUP MEMBERS

B.1 Matthew Dinh

Worked on some of the dataset preprocessing such as feature presence and feature count. Also worked on a couple techniques for feature elimination such as a meta-transformer using LassoCv and removing features with low variance. Also participated in the writing of this paper.

B.2 Samuel Mauck

Created the SteamDB web scraper, constructed the dataset, set up the Colab notebook, participated in paper writing, performed data preprocessing on time-series features, and experimented with programmatic feature elimination, and trained ensemble and graident boosting regressors.

B.3 Chase Dudas

Dataset preprocessing, converted categorical features into value vectors, participated in writing this paper, researched future work, PowerPoint Layout.

B.4 Connor Cook

Transformed the categorical features into one-hot vectors, set up LaTeX paper formatting, participated in paper writing, set up basic normalization and feature reduction (via PCA) portions of the pipeline, and trained some regressors.

Matthew Dinh - CSCI 4502
Samuel Mauck - CSCI 5502
Chase Dudas - CSCI 4502
Connor Cook - CSCI 5502B

An Analysis of Factors that Contribute to a Successful Steam Game

The key results of this project was a series of different regressors meant to predict the success of Steam games based on their static and dynamic features. The dataset was constructed from a combination of the official Steam API and the third party website SteamDB. The best regressor was a gradient boosting regressor, which an R^2 value of 0.47. The results of the other regressors are shown below:

Model score on test set

Model	Score (R-squared)
Linear Regressor w/ PCA	0.28
Support Vector Machine w/ PCA	0.09
Support Vector Machine w/ RFE and MAS	0.28
Voting Ensemble	0.38
Gradient Boosting Regressor	0.47

Figure 1: This table shows the results for different models.

In addition, the relative feature importance was extracted from the gradient boosting regressor, and plotted below:

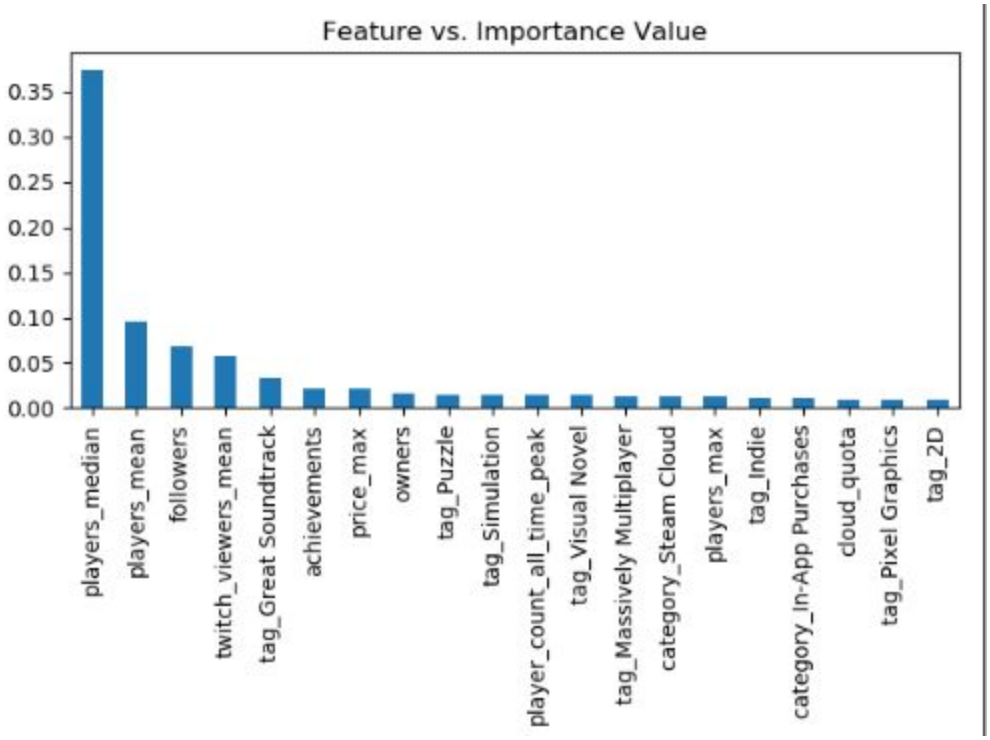

Figure 2: This chart shows the importance of the top 20 features.

Finally, when performing recursive feature elimination, the optimal number of features for this problem was determined to be 448, as depicted below:

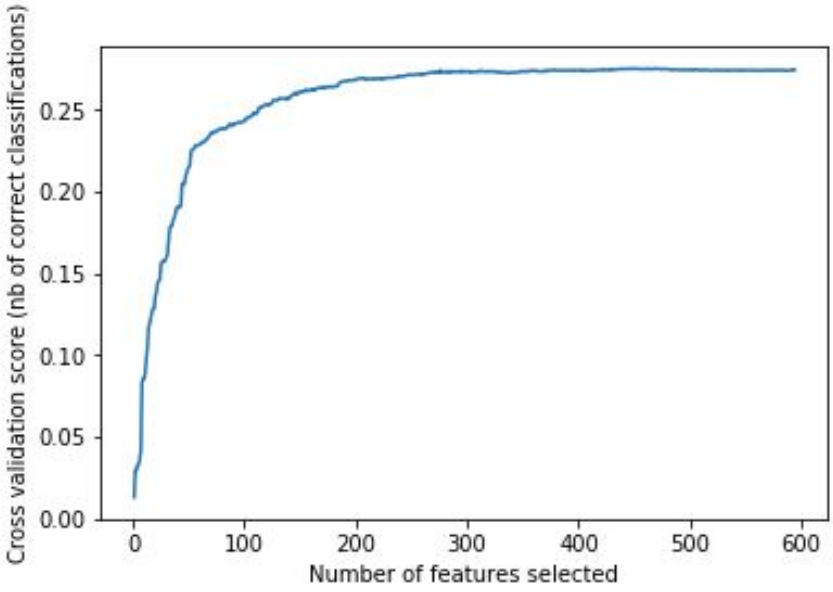

Figure 3: This chart shows the score for the linear SVM model with different numbers of features selected.