Predicting Game Popularity from Steam Descriptions

Joseph Liu

Abstract

In many cases, game descriptions are some of the first places where potential players learn about games. Therefore, it is imperative that publishers and developers write interesting descriptions that positively impact sales. In this project, we investigate the correlation between game descriptions and game popularity, independent from gameplay, using various models. We begin with a classification problem, including our baseline, Softmax Regression and our best model, Bidirectional RNN, and also experiment with different data representations and eventually regression. We conclude that while much of a game's popularity is associated with gameplay, the description also has a non-negligible impact on popularity.
Code Link:
https://github.com/jliu7350/csci467_final_project

1. Introduction

The game industry has grown significantly since the advents of PCs, or personal computers, and smartphones. In this study, we will investigate the popularity of PC games specifically and its relation to game descriptions. This problem is especially significant for smaller indie developers who have a smaller marketing budget. A good description will naturally draw more players, increasing profits with few additional resources. The ability to get feedback on draft descriptions could help developers retain players that stumble upon the store page - a valuable source of income for those who can't afford much publicity.
Steam is the world's largest game distribution service, and is the most straightforward location to find data. For this study, we use the descriptions that games provide on their Steam page as our dataset, and the number of reviews that a game has as a measure of popularity. Using these, we build a model to output the popularity of a game given its natural language/text description as input. We utilize a variety of techniques, including both classification and regression. We begin with a softmax classifier as our baseline, with labels created by binning our popularity measure and representing inputs in a bag-of-words format. We additionally experiment with Naive Bayes and Support Vector Machines with a similar setup. Lastly, we implement a Recurrent Neural Network using word vectors as input. For this model, we tried classification, with the same labels as before, and experimented with regression. Our work shows that while there is some correlation between the description and popularity, much of a game's popularity is dependent on additional external factors.

2. Related Work

Much related work has been done in analyzing game popularity. Previous efforts include game tags (Zhang et al., 2020) and more commonly, review sentiment (Ji, 2019; Zuo, 2018). It was found that throughout the epidemic, mentions of COVID-19 in reviews rose, along with the popularity of adventure games. It was proposed that this was a result of quarantine measures, especially in China. This analysis was conducted with Linear Regression, Support Vector Machines, and Decision Trees. In all three cases, the model was trained to predict differences in ownership based on weekly data, and then choosing the tags of the games with the greatest difference. This work was quite different, as the dataset was primarily based on COVID information rather than details about the game itself.
Additionally, sentiment analysis was done on reviews, and results of 70-80% accuracy were achieved in classifying reviews as either negative or positive, across both balanced and unbalanced datasets. Zuo's work, for example, was done with Naive Bayes and Decision Trees. The goal was to predict the sentiment (positive or negative) given the text in a review. However, no work regarding the popularity of games specifically, nor the use of the game description as a dataset, was found. As this field seems to be relatively niche, there is significant potential for future work.

3. Dataset and Evaluation

For our dataset, we have scraped Steam's website for the description. The description plaintext was located with the `game_area_description` HTML id and extracted using BeautifulSoup4 [1]. Additionally, we have found a dataset

[1] https://www.crummy.com/software/BeautifulSoup/

on Kaggle, the "Steam games complete dataset" [2], with supplementary information including review counts, game names, and tags. After filtering out non-game content (e.g. DLCs or other apps), we have a total of 17363 games. While the complete dataset has 21 columns, the only relevant features are `url`, `name`, `types`, `all_reviews`, `developer`, `publisher`, `game_description`.

We have also preprocessed the data, extracting certain features. Notably, we have extracted the total number of reviews from `all_reviews` and the ID from `url`.

We will now briefly explain how we use each of the relevant features. The game name, developer name, and publisher name features are all used as blacklists. In particular, large game studios (e.g. Ubisoft or Bethesda) have names that often show up in descriptions of their games. Any occurrence of this text in our dataset, especially in a bag-of-words representation, will cause overfitting due to their games' popularities. Additionally, due to the number of games that large companies release, it is possible for these to enter our final vocabulary for bag-of-words, even if we limit it to the few thousand most common words. We remove these words from the token lists for each description to prevent this. Next, the number of total reviews was extracted from the `all_reviews` column, which is plaintext. This is also used to generate an additional column, the class of each game (more details on labels later). Lastly, the type was used to filter out games from Downloadable Content (DLCs, or expansion packs), and the url was used to download the game description through web scraping.

The description itself also required processing. We started by tokenizing the words, by stripping all non-alphanumerical characters and converting the string to lowercase. We then filter out names, as discussed above, and use NLTK [3] to tokenize the words. Them we use the NLTK stopword corpus to remove common but useless words, such as "a", "the", or "and". From here, we create two separate datasets formats.

The first is a bag-of-words representation, used for our simpler models. To do this, we first stem our tokens, also using NLTK. After this, we use scikit-learn's[4] `CountVectorizer` to convert the list of tokens to a bag-of-words representation. Notably, this utility allows a max feature limit, which would return the n most common words. We vary this value as a hyperparameter, which we will discuss later.

The second format is word vectors, specifically GloVe [5]. GloVe is similar to word2vec, in that it measures word co-occurrence. The main difference is that GloVe does this globally, while word2vec does it locally. To get these embeddings, we use a pretrained mapping, `glove-wiki-gigaword-50`, provided by the gensim [6] Python package. This mapping was chosen out of all the mappings provided by gensim (including other GloVe and word2vec mappings) because of two reasons. First, its data source includes Wikipedia, which is a much more "balanced" dataset compared to Google News. Articles containing the words "World War 2", for example, often carry (and rightly so) negative connotations in news articles. However, for the purposes of games, it is used in describing historical settings. Wikipedia is much more balanced in that there are many articles that describe World War 2 that focus on historical events, like battles. These pages carry more associations with the setting and the historical backdrop that these events are occurring in than the emotions you may find in news articles. Our second reason is that we want an embedding that is relatively small. Our chosen mapping is 50-dim, while word2vec is 300-dim. This difference is important, as due to the relatively small size of our dataset, there are most probably multiple words that appear only once or twice. With large embeddings there are significantly more parameters, leading to potential overfitting. Using this information, for each description, we take the token list from before (with stopwords removed) and replace each token with its embedding from gensim. A total of 24 empty lists are removed, for a total of 17339 remaining games for this dataset.

Since we are using both classification and regression, we need to create labels that can accommodate both. For our regression case, we use the number of reviews a game has as our objective. The problem with this approach is the imbalance of data: 60% of our data has under 107 reviews, and the top 5% have more than 22000 each. Should we train on this data directly, the model would learn to give every game a small number, because that's where the vast majority are. To counteract this, we note that we actually don't care about how many reviews a game has, only how many it has relative to other games. A game with 1000 reviews in a vacuum (that is, without knowledge of how many players play games in general) is neither popular nor unpopular. Therefore, we redistribute the data using $y = log(log(y_{actual})) - 1.45$. This has the effect of significantly flattening the original spike in the data, as well as somewhat normalizing it to be between -1 and 1 with a mean at $0.0009 \approx 0$.

For the classification task, we must divide the data into distinct classes. We start by splitting the games into five roughly equally-sized classes, ordered by number of reviews. The classes were, in increasing order, $0-18, 19-39, 40-107, 108-445$, and $446+$ reviews, with $3427, 3439, 3521, 3498, 3448$ data points respectively. The boundaries of these classes, can be seen in Figure

[2] https://www.kaggle.com/datasets/trolukovich/steam-games-complete-dataset

[3] https://www.nltk.org/

[4] https://scikit-learn.org/stable/

[5] https://nlp.stanford.edu/projects/glove/

[6] https://radimrehurek.com/gensim/

1, overlaid on the recalculated distribution. Note that these numbers were selected purely for creating relatively balanced classes. We index the classes in increasing order (that is, class 4 has the most reviews and class 0 the fewest), and will refer to these in later sections.

We also split the data into train/test/dev sets with a

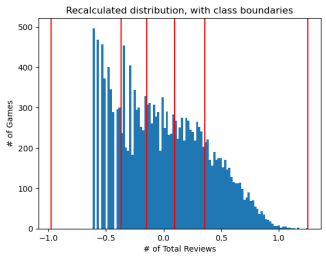

Figure 1. Distribution of games, after redistribution, with class boundaries drawn in red.

$70/15/15\%$ ratio. As our dataset is relatively large, we can afford to have proportionally smaller dev and test sets. Our resulting bag-of-words datasets have 12154, 2605, and 2604 for the train, dev, and test sets respectively, with the samples from each class roughly balanced between them. Our word vector datasets have 12137, 2601, and 2601 respectively.

Due to the fact that our classification dataset is relatively balanced, we can use accuracy as a simple metric. We will also examine confusion matrices to determine model performance. While precision and recall are still possible, due to the multi-class nature of our work, we would need one number per class which is much harder to interpret. For regression, we will simply use Mean Squared Error (MSE). Additionally, we will bin our outputs in the same way as the original dataset, to compare those results with the classification models.

4. Methods

We begin by framing this as a classification problem, with the description as input and the popularity as output. An intuitive way to measure popularity is by number of reviews. A game with few reviews is clearly unpopular, and a game with many positive reviews is clearly popular. However, a game with many negative reviews is actually interesting as well: It means that many players have tried it, and hence the description itself is attractive. As such, we choose this as our target.

As our baseline, we use a simple softmax classifier with bag-of-words as input. That is, we count the number of occurrences of each word in a preset dictionary (the top n most common tokens in our entire dataset, excluding stopwords) and convert the counts to a vector, without accounting for word position. This vector is then used as our input. The length of the vector, n is determined by our chosen dictionary size. As we have already removed stopwords and stemmed tokens, the majority of the remaining words will be relatively information dense. However, we also do not wish to have overfitting due to recognizing certain rare words. Therefore, we will experiment with different dictionary sizes as a hyperparameter.

After finding the input vectors, we pass them into our softmax model, implemented with scikit-learn. Our chosen solver is Stochastic Average Gradient descent, which is the most similar to the gradient descent covered in class. Specifically, during training, it selects one sample at random and updates its gradient only, instead of computing the gradients of all examples. The gradients of all other samples are kept from the previous iteration. That is, when calculating the gradient, it computes $\nabla f_i(w)$ for exactly one i, where $\nabla f(w) = \sum_i f_i(w)$. This is opposed to standard Stochastic Gradient descent, where $\nabla f_i(w)$ is computed for all i in the batch. The update rule remains the same, as does evaluation: $p(y = 1|x) = \frac{exp(w^{(j)T}x)}{\sum_{k=1}^{c} exp(w^{(k)T}x}$, as per standard softmax regression. The scikit-learn implementation additionally implements L2 regularization. The maximum of the resulting vector is then used as our prediction. Logistic Regression has few parameters, and as this is our baseline, the only hyperparameter we change is `max_iter`, which increases the number of iterations training can run for. This is done to prevent a convergence error.

Our second model is Naive Bayes with Laplace Smoothing. The scikit-learn implementation is standard, and the same as what was covered in lecture. Specifically, we use the `MultinomialNB` class. As from lecture, we have:

$$p(x_j = u|y = k) = \frac{count(x_j = u, y = k) + \lambda}{\sum_{i=1}^{n} 1[y^{(i)} = k]d_i + |v|\lambda}$$

Our input vector is evaluated with this equation, and the class with the maximum p is returned. Naive Bayes has no hyperparameters except for the smoothing parameter. While we experiment with various values to account for the sparse nature of our input matrix, it seems to have little impact (see Figure 2), so we use the default value of 1.

Our third model is the Support Vector Machine. As scikit-learn offers many implementations, we have spent some time to research them. Ultimately, we chose to use the `LinearSVC` class. Experimenting with `SVC` shows that due to the number of features and quadratic runtime, it takes significantly longer but does not seem to show much improvement and wasn't covered in class (the Linear SVM implementation was). Therefore, we chose the Linear SVM. It does not use a kernel, and thus is minimizing the function $\sum_{i=1}^{n} max(0, 1 - y_i(w^T x_i + b)) + \lambda||w||^2$. Note that the scikit-learn implementation includes L2 loss. Different from lecture, however, it uses Squared Hinge Loss. That is, the

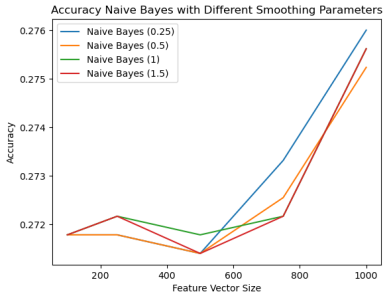

Figure 2. Naive Bayes performance on dev set with different smoothing parameters

summation has the form $\sum_{i=1}^{n} max(0, 1 - y_i(w^T x_i + b))^2$. All other implementation details seem to be identical to what was covered in class. Our binary prediction is ultimately computed by $sign(w^T x + b)$. The scikit-learn implementation has one such classifier for each class, where positive signals the class and negative signals any of the other classes. The maximum of the classifiers (argmax) is taken as the final output.

In all three cases above, we use the bag-of-words input. This is passed through the chosen model, p is computed, and the maximum is chosen as the model's prediction. All hyperparameter tuning (for all models) is done by training on the train set and evaluating on the dev set.

Our last model is a Recurrent Neural Network. Clearly, the bag-of-words input that we have used thus far won't work with RNNs, and we therefore switch to word vectors. Our RNN architecture utilizes bidirectional Gated Recurrent Units, or GRUs, as discussed in lecture. That is, there are two distinct RNNs, with the reversed input passing through one of them, one word vector per time step, and the regular input passing through the other, again with one word vector per time step (Figure 3). The outputs of the two are concatenated, and passed through a ReLU (for non-linearity) and Dropout layer (randomly removing connections to prevent overfitting). The final linear layer has either five neurons, for classification, or one neuron, for regression. This architecture and data representation, as mentioned previously, will provide much more information about context and word meaning than a simple bag-of-words model. As this was our main problem with bag-of-words (see sections 5 and 6), our new architecture will allow us to find more detailed patterns about word usage. Now, we will discuss each of the various layers as well as the hyperparameters in them.

The first layer each input reaches is the GRU. As the GRU is a variant of an RNN node, it has "time steps", with a hidden state persisting through it. The input for each time step in our case is simply the next word vector in the input. Given this input x_t, the GRU first computes $r_t = \sigma(W_{ir}x_t + b_{ir} + W_{hr}h_{(t-1)} + b_{hr})$ and $z_t = \sigma(W_{iz}x_t + b_{iz} + W_{hz}h_{(t-1)} + b_{hz})$. r_t is the reset gate, and essentially decides how much past information to discard. The higher it is, the more information from the past it preserves into the output. z_t, on the other hand, is the update gate, which decides how much of the previous hidden state should be copied forward, as opposed to the new information. That is, if $z_t = 1$, then it carries over all of the past information and ignores the new input. After this, it computes $n_t = tanh(W_{in}x_t + b_{in} + r_t * (W_{hn}h_{(t-1)} + b_{hn}))$, or the new gate. This is the new hidden state, which is then modified by the update gate to get the final output $h_t = (1 - z_t) * n_t + z_t * h_{(t-1)}$. Combined, this allows us to remember states longer. The hyperparameters we tune here are the dropout probability (p=0.5, between each layer of the GRU), the hidden state size, and the number of layers. For the hidden state size and number of layers, we search a variety of options and graph the losses incurred by each (see Experiments).

After passing through the GRU, the two last hidden states (forwads and backwards) are concatenated and passed through a ReLU and Dropout, as mentioned earlier. The only parameter is with dropout, and we use p=0.5. Lastly, in the Linear layer, we change the number of neurons depending on whether we are doing classification or regression. For classification, we have five neurons in a linear layer ($y = xA^\top + b$), followed by a log-softmax ($LogSoftmax(x_i) = log(\frac{e^{x_i}}{\Sigma_j e^{x_j}})$). Then, we take the negative log-likelihood loss ($-x_{n,y_n}$ where x is the input, y is the target) relative to the actual class of the datapoint. We also experiment with regression, for which we simply take the output of the final neuron and calculate its error (Mean Squared Error, specifically) with respect to the redistribution. All of this is implemented using the standard Pytorch layers, and all the equations come from the documentation on the Pytorch website also.[7]

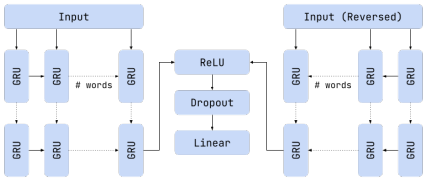

Figure 3. RNN Model Architecture

5. Experiments

We will now discuss the results of each of our models. Figure 4 (below) displays the accuracy of the bag-of-words models (the baseline, Naive Bayes, and SVM) over various

[7] https://pytorch.org/docs/stable/index.html

feature vector sizes on the dev set. We also include a fourth line at ≈ 0.2027 representing the best possible prediction by just picking class 2 always.

We notice that all these models were significantly better

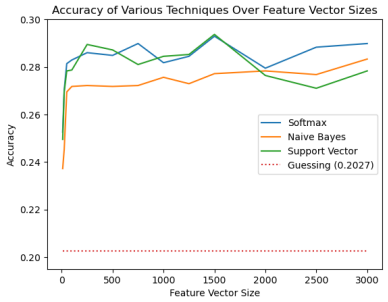

Figure 4. Dev Accuracy of Models by Feature Vector Size

than guessing, and that improvement is very fast as the number of features increases initially but flattens out quickly as well. We also note that softmax regression (in blue) performs surprisingly well compared to the other two techniques, despite being the baseline. By contrast, Naive Bayes performed poorly, possibly due to feature dependence. For all three models, however, we find a peak at a feature vector size of $\approx$ 1500, so we evaluate the model on the test set and create confusion matrices there (Figure 5). The accuracies are also in Table 1.

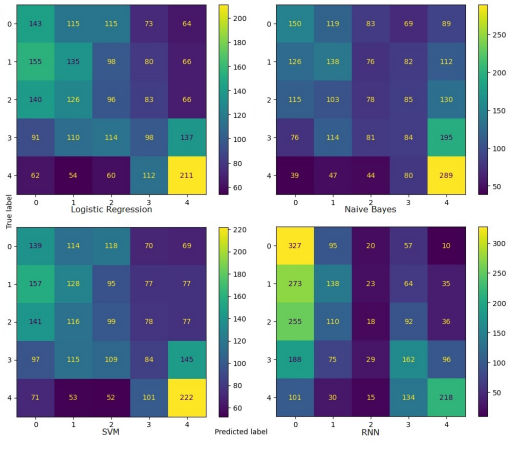

Figure 5. Confusion Matrices of Classification Models on Test Set

Surprisingly, Naive Bayes did better than the other two models on the testing set. This could be explained by the variation seen in accuracy on the dev set. Clearly, there are large amounts of oscillation, with accuracy both falling and rising quite drastically. Therefore, 1500 might be a point where Softmax and SVM happen to have good performance on the dev set, with Naive Bayes happening to have a worse performance.

Model	Accuracy on Test Set
Logistic Regression	26.23%
Naive Bayes	28.38%
SVM	25.81%
RNN (Classification)	31.99%
RNN (Regression into Binning)	23.57%
Random Choice	20.03%

Table 1. Peak accuracies of each model

Now, we will turn our attention to our RNN model. We first find the hyperparameters, testing out various combinations for different numbers of hidden layers and the size of the hidden state. While batching is a possibility, we found that the highly varying input lengths resulted in unnecessary complexity, so we chose a simpler batch size of 1. For this training, we use 20 epochs at a relatively high learning rate of 0.0005. As this is an exploration of the hyperparameter space, we decided to sacrifice some accuracy in order to explore more combinations. From these experiments, it is shown that the model with 16-dim hidden state and 3 layers has the lowest loss ($\approx$ 1.5150) on the dev set at epoch 9 (shown in Figure 6), and we therefore take that as our final model. On the test set, we were able to achieve an accuracy of 31.99% with this model. We also see that it tends to favor the two extremes (See figure 5, bottom-right), which shows that the model is actually identifying relevant patterns. Lastly, we test out regression with the similar architecture described in section 4. While it is not the main focus, we notice that generally, the upper-left and lower-right corners of all the confusion matrices are brighter than the other two, even if the classifier got the answer wrong. Therefore, we decided to conduct a preliminary exploration of this area.

To convert this problem a regression problem, it suffices

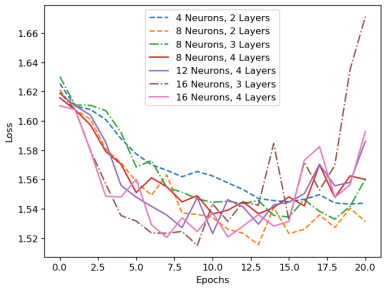

Figure 6. Loss Curves on Dev Set by RNN Hyperparameters

to replace the linear layer from the classifier with a single neuron, which is our output. From this, we can simply use Mean Squared Error, which has the value $mean_n((x_n - y_n)^2)$. Training a model with the exact same parameters (learning rate = 0.0005, 16-dim hidden state, 3 layers) as our "best" classification model results in an

underwhelming model. We generate a confusion matrix by binning the outputs in the same way as our inputs for the classification task (Figure 7). While we have a similar vaguely-diagonal pattern as before, the regression model is basically always just guessing the middle class and thus doesn't perform particularly well. Therefore, we choose not to explore this for the time being.

Overall, we find that our models are not that accurate,

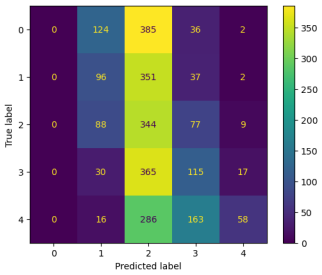

Figure 7. Confusion Matrix of Regression RNN on Dev Set

regardless of data representation. Of course bag-of-words has worse performance: By removing word contexts, it makes it hard for the model to recognize what each word means relative to the rest of the text. With many games, both good and bad, having the same theme or setting, this makes many features limited in use. Thus, it is possible that all those models are limited by features, and produce similar results. However, while the word vectors/RNN combination does slightly better, the improvement isn't too drastic. Therefore, the primary limiting factor seems something unrelated to our data representation.

6. Discussion

As our models got so many examples wrong, we will do a general analysis of the results. First, from the confusion matrices, it is evident that the model has found some correlation between features and popularity. As noted before, lower-left and upper-right sides of the matrices are darker in all matrices. This shows that even when the model does get it wrong, it's not wrong by too much - it usually has some idea of where it should go. As a result of this, we conducted a preliminary investigation of regression. However, the lacking results of regression suggests that there is something else at play.

Secondly, we investigate the most impactful features in our bag-of-words models. In particular, "franchis" is very impactful, being in the top 5 most important features in multiple models. This makes sense, as franchises typically are more successful. On the other hand, common words like "game" are among the least impactful, due to their commonality in descriptions. Additionally, none of the factors seem to be overfitting, as all the weights seem reasonable. From this, we can conclude that one way to increase accuracy is to remove dataset-specific common words instead of just generally common words, but this will still be limited by the representation.

Lastly, we will investigate cases where our RNN model performed very poorly. A few games where the RNN classified 4 (high) when it's actually 0 (low) are NetHack: Legacy, Dual Blades, and Blackjack In Space. While it is difficult to explain a neural network exactly, there are a few possible explanations for some of its decisions. Blackjack, for example, is a really popular game, and its word vector probably reflects this. Dual Blades' description contains the names of other very popular games, such as Street Fighter, while NetHack: Legacy is a "remaster" of NetHack. Again, we know that words like "remaster" typically are good indicators of popularity. Overall, it will be much harder to "fix" some of these errors, as they are valid features that other games may have. To understand when a "remaster" is something that's really hard to predict, and depends on the original game. However, we can't leak that information, as it will make it too obvious what the current game is. It's possible that a Transformer architecture could manage this, and this is an interesting direction for future work.

In other descriptions, however, it seems like there isn't a reason for the game to not be popular. Indeed, the dataset we are working with is, by its very nature, quite "noisy" with respect to the description. A game can have no description at all and still win Game of the Year. Ultimately, it's the game itself, and not the description, that is the greatest factor in game popularity. In that sense, our model has already performed quite well, and beyond expectations: It has demonstrated that descriptions have a non-negligible impact on the popularity of the game.

7. Conclusion

In this project, we have used a variety of models to predict game popularity from natural language descriptions. We have explored the use of different classification models, including a baseline softmax regression, Naive Bayes, SVMs, and the best-performing Bidirectional GRU. Ultimately, however, changing the data representation, reframing the task as a regression problem, and the many other tweaks did not have a very significant impact on our performance. It seems as though there are factors independent of what we analyzed that impact our objective in a more significant way than game descriptions. Ultimately, this project has taught us that models are only as good as the data they are given, and when there are few correlations to be found, even the most complex of models will still have a limit.

References

Ji, F. Sentiment analysis and opinion extraction of game reviews on steam. 2019.

Zhang, J., Zhao, H., Chen, Z., Song, Z., et al. Prediction of the most popular game tags on steam under the influence of covid-19 based on machine learning and natural language processing. *The Frontiers of Society, Science and Technology*, 2(12), 2020.

Zuo, Z. Sentiment analysis of steam review datasets using naive bayes and decision tree classifier. 2018.