Sentiment Analysis of Steam Reviews

By: Ben Strumeyer
&
Bikram Ghosh Dastidar

1. Introduction

Steam is a well-known gaming platform downloadable for anyone who has a computer. The Steam Database, reachable at http://store.steampowered.com/ is a website containing information on all of the games that the platform has such as what controllers are supported for the game, how much the game costs, and even user reviews for each game on the store. Our project idea was based on the latter, the thousands of reviews that each of individual games have.

The idea is to analyze the sentiment of these reviews written by the steam community. Our goal was to computationally predict whether a review was recommended or not recommended by the reviewer, solely based on the review text body itself. For example, if a review text is, "Terrible game. I wouldn't buy this because the controls are too confusing," we would hopefully predict this reviewer to rate this game as, "Not Recommended," because of phrases such as, *"Terrible Game," "wouldn't buy,"* and *"too confusing."*

In hopes of realizing this project, we aim to understand the process of sentiment analysis. How is the review text parsed? Are there certain phrases that give better indications of whether a review is recommended? How do we use these phrases to predict whether or not a review recommended? Can our method of sentiment analysis accurately analyze and predict the lingo of steam reviewers, since their wording is non-standard and doesn't provide conventional styles of speech like in books. Our goal is to find answers to all of these questions.

2. System Description

Our system is a web application with the server being the part that does the heavy loading of parsing, analyzing and annotating reviews both during training and testing. Our client side is merely a UI that cleanly shows the results of the system in real-time. The client shows the training and test results for each game, the accuracy at which the system performs for that particular game, and the top few positive / negative things that people have talked about for the game. The client also has functionality to browse the entire library of reviews for each game, annotated by the phrases extracted by our system, which are respectively colored according to their polarity (green for positive, red for negative etc.).

We chose NodeJS to be the technology that our server would be built with. We wrote the application in TypeScript, Microsoft's version of augmented JavaScript. The reason we chose NodeJS was because we were aware of the performance requirements for such a complex application. Since NodeJS is asynchronous by default, we harnessed that potential to delegate time consuming tasks, like training and testing, to

asynchronous workers that could simultaneously process enormous chunks of data. We used MongoDB as our database, since we wanted our collection of data to be flexible and non-relational.

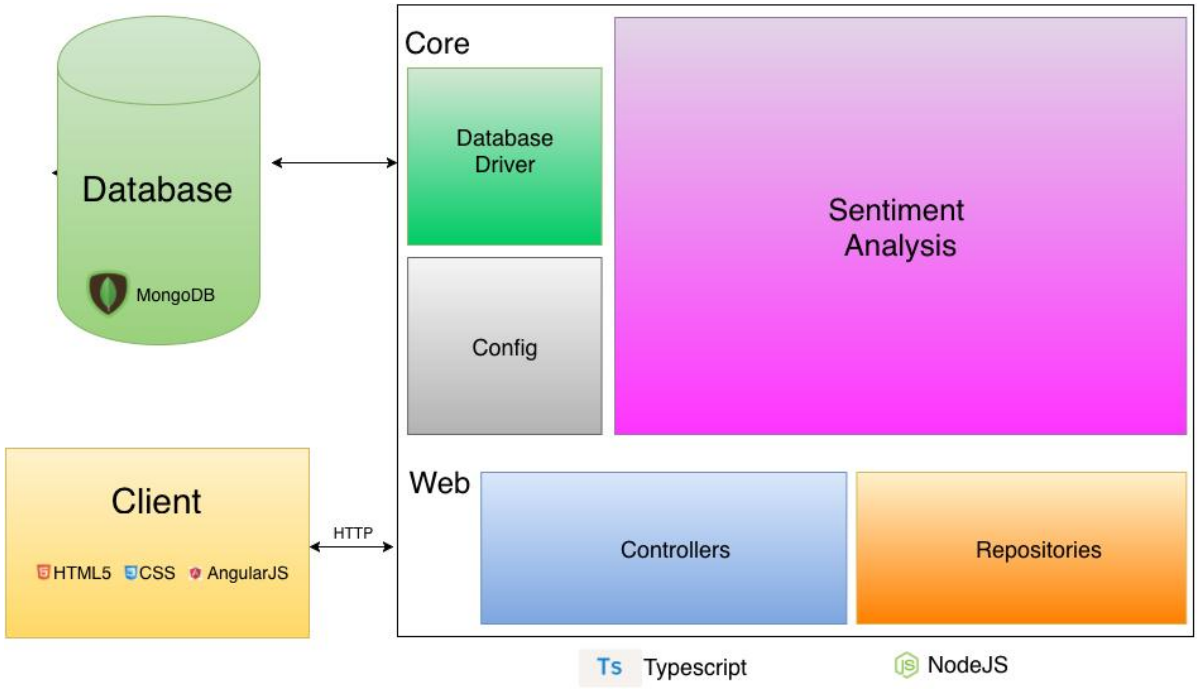

Fig 2.1 – System Architecture

2.1 Gathering Data

We implemented a Scraper and a Parser that scrapes and parses the Steam website to gather information about reviews. We picked a culmination of all of the reviews of 10 different test games. Each review contains a review body which is the text we parse. In addition, we also take as input whether the author recommended it or not. This allows us to train our system and compute the accuracy of our results. We go through almost all reviews for the games we picked to test our system on, and then save this information into our database.

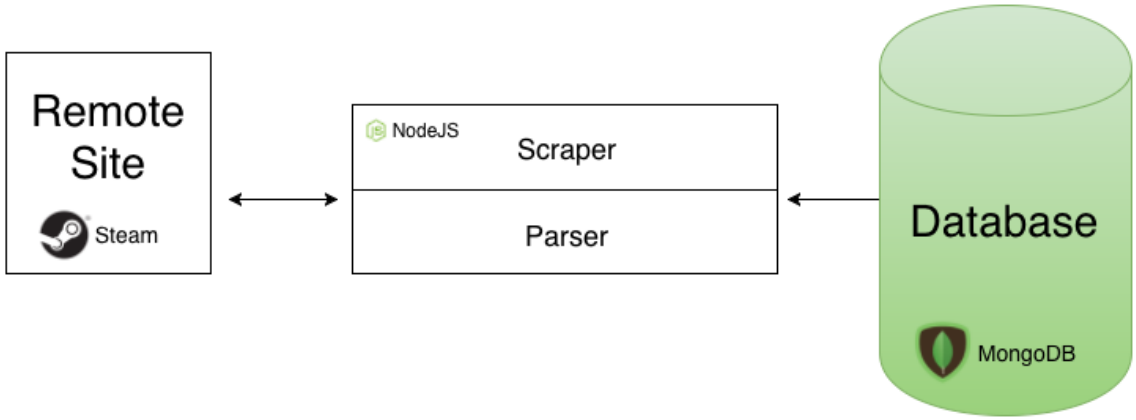

Fig 2.2 – Scraper / Parser

2.2 Training

During training, we take the data exported by our scraper/parser and we send it through a POS tagger. After each review body is tagged with parts-of-speech tags, we send it through a Phrase Extractor to extract relevant phrases that express sentiment. At this point, all we do is save these extracted phrases to the database.

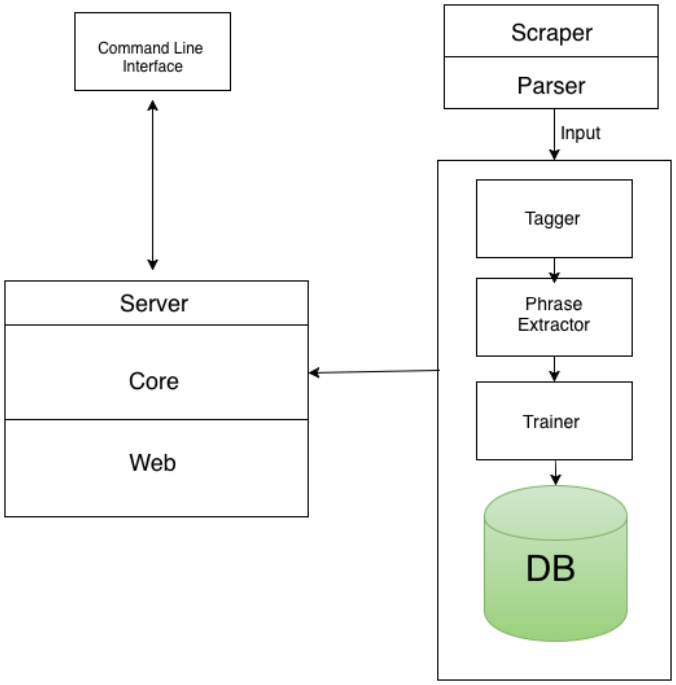

Fig 2.3 – Training

2.3 Testing

We realized that testing the system with a game that we also trained on made no sense. So we implemented logic in our system to use training data from *all other games* besides the game we are testing on as the training set. This way we can get accurate values of our system's performance.

During testing, we take a game's App ID as input. This gives us all the reviews that are associated with the game. We send these reviews through the POS tagger and the Phrase Extractor like we do in training. After that step, we send each review to a Sentiment Analyzer which checks against the database and does computations on phrases it encounters to give us polarity values for each phrase. After each review is tested, we save the corresponding test results back into the database for retrieval later. When the client requests these reviews, we send the individual reviews back to the client along with the phrases and polarities for each reviews. The client uses this information to show a collection of annotated positive and negative reviews.

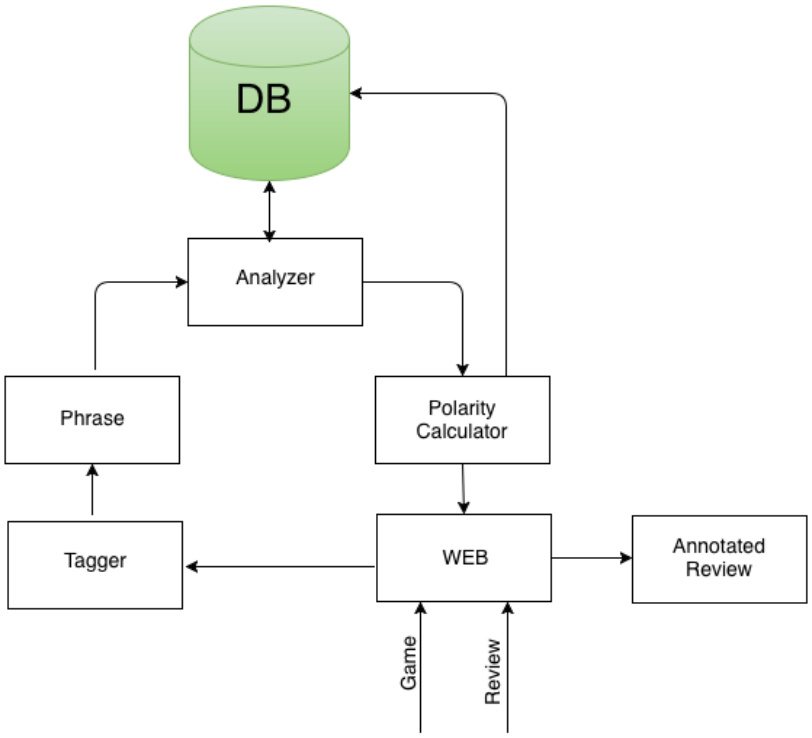

Fig 2.4 – Testing

2.3 Evaluation

After both training and testing, we run our Evaluator through each game to compute different statistics like accuracy, top 10 good/bad things about a game, total percentage of positive/negative reviews etc. This evaluated data is available to be displayed on the client in real-time, since both the analyzer and client point to the same database.

3. NLP Technique and Algorithm

We used our own modification of the Turney Algorithm in order to accomplish this sentiment analysis task. It's an algorithm created by Peter Turney in 2002, that's based on polarity similarity of word pairs discovered by Hatzivassiloglou & McKeown in 1997. For example, words such as "nice," and "fair," have the same connotations, and have very low polarities, whereas "fair," and "corrupt," have very high polarities because the words have overwhelmingly semantic differences.

An overview of the Turney algorithm has three steps. We start off by extracting a *phrasal lexicon* from reviews, which are bigrams based off of specific trigram linguistic rules shown in the second table below. We then learn the polarities of the phrases, and finally compute the average polarity of a single review to predict whether it's recommended or not.

The first step of the algorithm requires the usage of an extraction method for our phrasal lexicon. We attempted to follow this strategy verbatim, shown below in table 3.1. Each row states a part-of-speech tagged trigram that the phrase in the review has to match. After doing so, the first and second word will be pushed into our database as a bigram.

First Word	**Second Word**	**Third Word (not extracted)**
JJ	NN or NNS	anything
RB, RBR, RBS	JJ	not NN, nor NNS
JJ	JJ	not NN nor NNS
NN or NNS	JJ	not NN nor NNs
RB, RBR, RBS	VB, VBD, VBN, VBG	anything

Table 3.1 – Turney's pattern of features for phrase extraction

After implementing this, we noticed that the phrase set size was very small. Our intuition led us to believe that it was due to the outlandish diction of Steam reviews, much like Twitter. Based on this observation, we decided to expand this set to contain all transformations of adjectives (comparative, superlative etc.) and all types of verbs (gerunds, past tense, participles etc.). This is shown in table 3.2 below, which gave us a much larger training set, and proved to be beneficial.

First Word	**Second Word**	**Third Word (not extracted)**
JJ, JJR, JJS	NN or NNS	anything
RB, RBR, or RBS	JJ, JJR, JJS	not NN nor NNS
JJ, JJR, JJS	JJ, JJR, JJS	not NN nor NNS
NN or NNS	JJ, JJR, JJS	not NN nor NNs
RB, RBR, RBS	VB, VBD, VBN, VBG, VBP, VBZ	anything

Table 3.2 – Our modified pattern of features for phrase extraction

After refining the training corpus to contain only matched phrases along with whether the phrase was in a recommended/not recommended review, we needed a way to compute the polarity of a phrase. The Turney algorithm utilizes the method below to calculate polarity. The input is two words with similar meanings to "excellent" and "poor," and states that positive phrases are more likely to co-occur within 10 words of, or NEAR the word "excellent", and negative phrases are more likely to co-occur NEAR the word "poor."

This co-occurrence is measured by obtaining pointwise mutual information, calculated by taking the ratio of a count of how many times the word co-occurs with a word such as "excellent" or "poor", divided by the count of how often the words occur independently.

$$PMI(word_1, word_2) = \frac{hits(word_1\ NEAR\ word_2)}{hits(word_1) * hits(word_2)}$$

Equation 3.3 – Pointwise Mutual Information

Finally, we obtain the polarity of a phrase by subtracting the PMI of a phrase, given "poor," from the PMI of the phrase, given "excellent" (or any excellent/poor synonyms of choice).

$$Polarity(phrase) = PMI(phrase, 'excellent') - PMI(phrase, 'poor')$$

Equation 3.4 – Polarity of a phrase using PMI

Simplifying the above, we get:

$$Polarity(phrase) = log_2\left(\frac{hits(phrase\ NEAR\ 'excellent') * hits('poor')}{hits(phrase\ NEAR\ 'poor') * hits('excellent')}\right)$$

Equation 3.5 – Simplified Polarity

A problem arose when we attempted to implement this algorithm. Turney used the NEAR function of the AltaVista search engine which allowed linear query time for phrases co-occurring NEAR "excellent" or "poor". Since we did not have access to such a search engine, our intuition led us to calculate NEAR by determining if a phrase merely existed in a review.

$$Polarity(phrase) = log_2\left(\frac{hits(phrase\ IN\ positive) * count(negative)}{hits(phrase\ IN\ negative) * count(positive)}\right)$$

Equation 3.6 – Our version of modified Polarity

Interestingly enough, this deemed the use of words such as “excellent” and “poor” obsolete, but still gave values shockingly similar to Turney’s original algorithm. Afterwards, calculating the average polarity of a review was trivial.

4. Evaluation

Our ideal output would return a correct prediction on whether the author recommended or did not recommend the review. Delving further into the Turney algorithm, a less ideal output is that we correctly extract all of the phrases in the review and we calculate the polarities of those phrases perfectly, but the review is wrongly marked.

The overall accuracy displayed in the table below, is a ratio of how many reviews we evaluated correctly, over the size of the set of reviews with respect to that specific game. This is computed for all games.

Game	Total Reviews	Predicted	Correct	Precision
Flatout 3	497	336	240	48.28973843%
Batman™: Arkham Knight	6870	5248	3666	53.36244541%
StarForge	6621	4464	3086	46.60927352%
Nether: Resurrected	9937	6776	4778	70.51357733%
Rocket League	34932	18118	12013	66.30422783%
Street Fighter V	3110	2325	1597	68.68817204%
Dragon Ball Xenoverse	5899	3724	2683	72.04618689%
School of Dragons	1421	782	541	69.18158567%
NBA 2K16	1733	1122	790	70.40998217%
The Culling	6074	3859	2783	72.11712878%

Equation 4.1 – Our system’s performance

We received precision ranging from 66%-72%, with Rocket League showing the worst performance. We believe this is due to Rocket League containing around 35,000 reviews, which is nearly half of our training set. Since we test each game on every other game, all games besides Rocket League tested on a training corpus of around 65,000-70,000 reviews. In comparison, Rocket League’s training corpus was only 40,000 reviews.

Furthermore, we noticed a trend that reviews with small lengths are pushed to the bottom of our review list when sorting by polarities. This is probably due to our phrase extractor having a low probability of matching phrases in smaller reviews, due to the smaller word count.

We also extracted the top 10 positive and negative "features" that reviewers have talked about in the reviews for each game. This shows us that the polarity values of phrases range from around 25.0 to -25.0.

Improvements

There were certain things that we planned but we didn't get enough time to implement them for this project. One thing we wanted to implement was multithreading which would have drastically improved our training and testing times. We attempted multithreading but simply did not have enough time to get it to work properly. Even though we split our workload into asynchronous workers, it would have been much faster if we used multiple physical CPUs.

We also wanted to fiddle with the phrasal lexicons and the feature set that we use to extract these phrases to see what optimized the system's performance. Since Steam's lingo is so absurdly different than normally occurring language, we wanted to see what syntactic structures they have that might optimize performance. We also wanted our system to pick up on sarcasm and repeated phrases such as "10/10 would play again".

Conclusion

From this project we learned how to do sentiment analysis on unstructured datasets from scratch. It was a challenging project but we had fun building our system. Going into the project we had no clue how to do sentiment analysis as it wasn't really mentioned in class. But as we figured it out gradually, we modified our application along they; even though we hit a couple road bumps, we managed to have a working system within the deadline.