

# Capstone Project Report

*By Martin Beierling as part of Udacity's Machine Learning Nanodegree*

Martin Beierling  
07/30/2017

## Definition

The project is named “Better Twitch Chat”. A live version can be found at <https://better-twitch-chat.firebaseio.com> and the source code is available at <https://github.com/mBeierl/Better-Twitch-Chat>.

## Project Overview

The overarching goal is to provide a better experience when reading a Twitch channel's chat. Twitch.tv is an online platform that allows to stream video and reach thousands of like-minded people on the internet. It is most commonly used to stream video of a videogame and a person playing the game. The chat is the main window of interaction with the streamer, but especially in channels with a lot of viewers the chat can become very hectic and unreadable by spam of messages that are of low value. An example of such a chat:



Here, one can clearly see the redundancy of certain messages. This project aims to fix this, by filtering out all unwanted messages. To achieve this, the project will use voting messages up or down to gather a dataset which will then be used to train an Artificial Neural Network.

## Problem Statement

We're standing in front of a binary classification problem:

"Should this message be hidden or shown?"

As mentioned before, the project will use an Artificial Neural Network (ANN) to solve this problem. The data is a list of messages that each have a boolean value of whether it should be shown or hidden. This data needs to be obtained, as it is not available publicly. For this, the project will implement a way to vote on messages in a live Twitch chat and save all voted messages to gather the dataset. To make sure that we don't gather a dataset that is biased toward one specific Twitch chat, we'll be gathering data from multiple Twitch channels.

After gathering the data, one problem is the representation of our text-messages. The project will use a model that maps text to an array of 0s and 1s, which is then ready to be used for training an ANN. This model's name is "bag-of-words", and allows us to use the occurrences of words in a message as a feature.

To finish the project, this trained ANN will be used to predict whether a new message should be shown or hidden. Using these predictions, the project will offer a better Twitch chat by only showing the messages the user wants to see.

## Metrics

It is important to find the best configuration for our ANN, as it will define the success or failure of the project in its aim to deliver a better Twitch chat. We will find the best configuration by hyper-tuning and testing the performance using Precision, Recall and the F1 score.

The F1 score uses Precision and Recall to define the performance of a model. The higher the F1 score the better. We're able to use this score in the project, as we're facing a binary classification problem.

These metrics are defined as follows:

$$\text{Precision} = \frac{tp}{tp + fp} \quad \text{Recall} = \frac{tp}{tp + fn}$$

$$F_1 = 2 \cdot \frac{1}{\frac{1}{\text{recall}} + \frac{1}{\text{precision}}} = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

Where *tp* means “true positives”, *fn* means “false negatives” and *fp* means “false positives”.

## Analysis

### Data Exploration & Visualization

As the dataset is not present beforehand, it needs to be gathered. This will be done using the method described in the Problem Statement section.

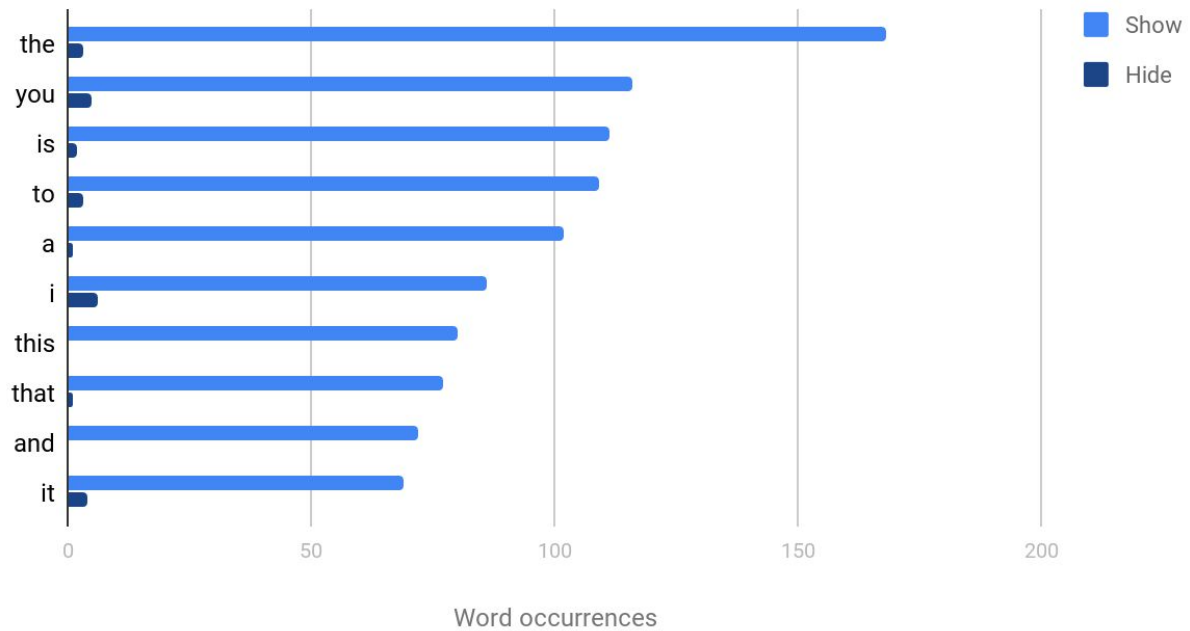
Yet we already know a lot about our data. We know that we’re dealing with text split into multiple instances, which are the messages sent by users. To represent our data using a bag-of-words model, we will need to create a dictionary containing all words present in all messages of our dataset.

The input space is rather big, as we will have hundreds to thousands of different words used. Nonetheless, messages are often composed of just a few repeating words and some rather rare words. This can also be seen e.g. in Shakespeare’s plays, where the most frequent words and “the, and, i, to, of, a, you, my, in”. More info about this can be found on this [statistics website](#).

After recording over 900 messages including thousands of words, I created word counts of messages that were voted to be shown and word counts of messages that should not be shown.

In messages voted to be shown, these are the top 10 mostly used words:

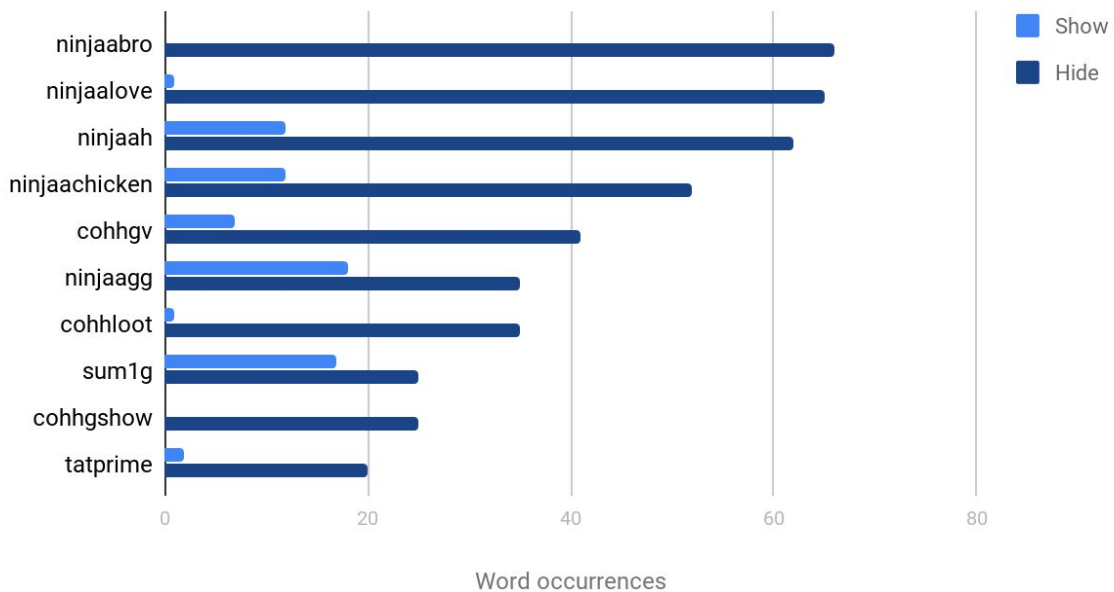
### Top 10 of "Show"



The significant information in this exploration: The top 10 word occurrences in to be shown messages have a very low count in the to be hidden messages.

Let's look at the top 10 mostly used words in to be hidden messages:

### Top 10 of "Hide"



---

Here, the difference between Show & Hide is slightly smaller than before. Another important lesson from this chart: The top 10 words in to be hidden messages contain Twitch channel specific words. As an example, the words prefixed with “ninja” are specific to the streamer [Anthony\\_Kongphan](#). These are also words that are converted to images in Twitch chat, as these represent specific emoticons bound to the channel.

As emoticons are part of the actual text-message, they will be taken into account. But as we will need to pay extra attention to these, as they may drive our model to be overfit for a specific Twitch channel. To overcome this issue, the long-term goal should be to gather as much data as possible from as many Twitch channels as possible.

Finally, as we found out in the first chart ‘Top 10 of “Show”’, the words of positive messages (those that should be shown) can be distinguished very well from words of negative messages (those that should be hidden). This is a strong indicator that the data should work well to train an ANN in our binary classification problem.

## Algorithms and Techniques

To feed our data into the ANN, we will need to transform the text messages into a more “machine readable” representation. For this, arrays work best and the domain of natural language processing has proven that the bag-of-words model [works very good](#). Even though we will lose the word context as additional feature, we will still have a lot of information inside of our word occurrence array, which the bag-of-words model essentially represents.

Furthermore, we will use an ANN (Artificial Neural Network) to eventually classify a previously unseen message as “show” or “hide”. More specifically, it will be a Feedforward neural network with backpropagation. I chose this Machine Learning model, because it is rather easy to use and produces good results. Also, there is a very good open source library ([brain.js](#)) to use ANN’s in the browser. Another reason is the rather large input space, as we will have thousands of words in our dictionary and every message will be mapped with a one-hot encoded array, representing the occurrence of a specific word. Neural Networks tend to [work very well with large dimensionality](#) of input data.

The project’s view and user interface will be built using web technologies, including React & Redux as well as Google’s Firebase as backend.

---

## Benchmark

Generally, the project should deliver a better Twitch chat. This can be decided upon by using the final web app of the project. As this is very subjective, I'd like to set an F1 score of 0.9 or higher to be the minimum score an ANN should deliver on our dataset.

It is also important to look at the individual precision and recall measurements. Precision should be rather high, as this defines whether the messages that are shown should actually be shown. On the other hand, recall shouldn't be too low, but it is not as important as precision, because recall basically describes how many messages were hidden by accident. To deliver a better Twitch chat, more focus should be put upon delivering a high precision than recall value, to achieve a really "clean" chat look.

---

## Methodology

### Data Preprocessing

Beside transforming the text messages into a bag-of-words model representation, no preprocessing will be done. Some theories that might lead to better results could be tested and are further explained in the results section.

The bag-of-words representation is achieved using the open source [mimir](#) library. It has a function to create a dictionary of all possible words of a given vocabulary and a function to create the actual bag-of-words representation of a given text based on a given dictionary. The result of the bag-of-words function is in the form of an array, where every index represents a word in the dictionary and the value at the index is the count of that word inside the text the bag-of-words is representing.

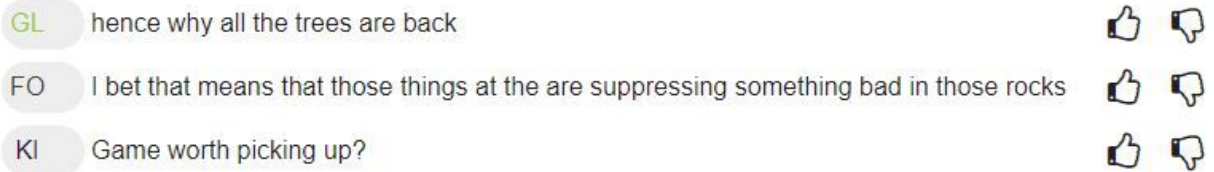
### Implementation

Implementation started with the Web App, which is mainly built using react, redux & firebase. The reason being that we first need an interface to see the chat and the ability to vote on messages in order to gather needed data.

As one can see from the commit history of [the project's GitHub repository](#), a fair amount of time has been spent on the actual front-end implementation. To create a good looking UI quickly, the [material-ui](#) library has been used. It is a set of React components that implement Google's Material Design. The reason for this is, that far less time had to be spent on actual UI & UX development, as the core elements like checkboxes, input-fields or menus are already well implemented by the material-ui library.

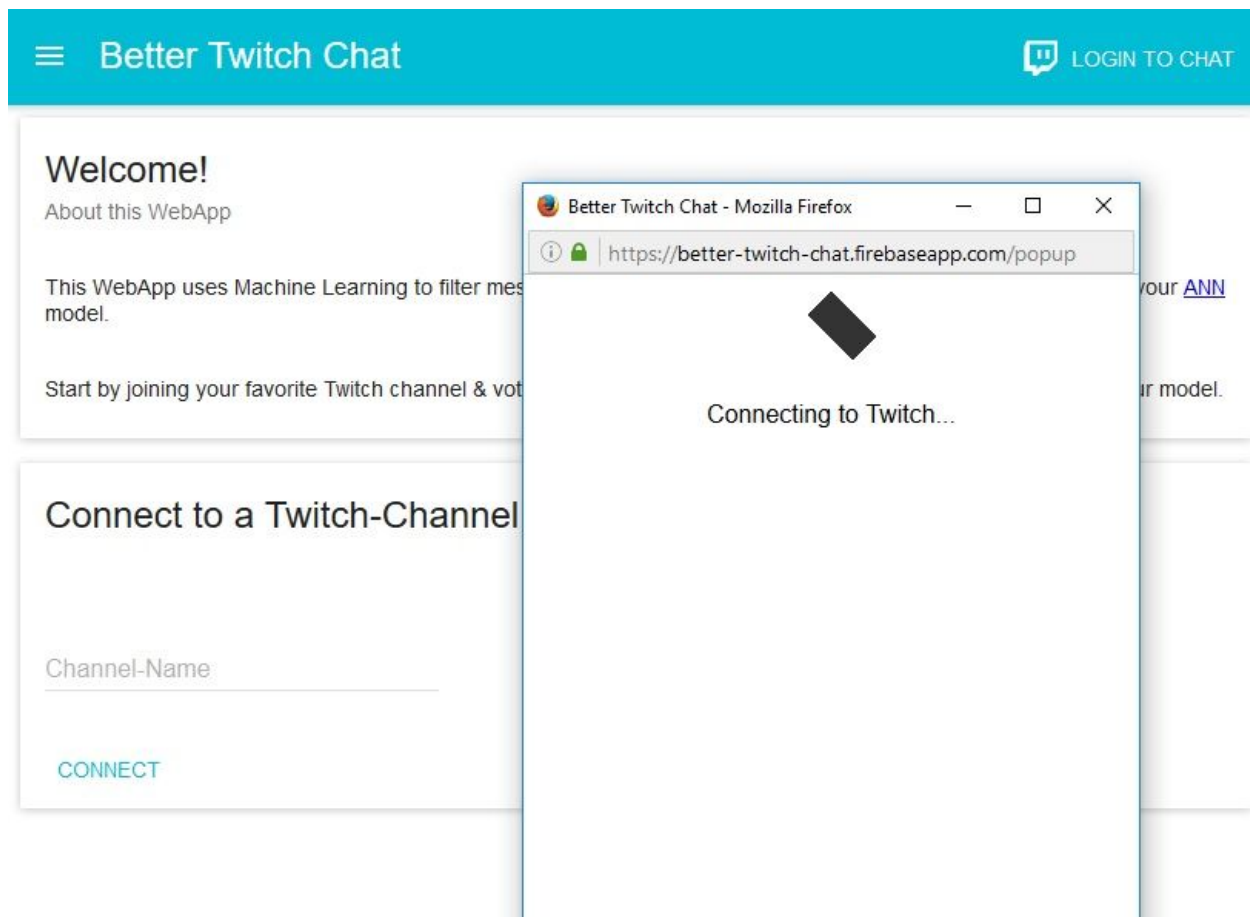
The first view that was finished is the actual chat view. It is the most important view, as it is used to display the filtered chat and interact to vote on chat messages to gather the dataset. Here is an image of how messages are displayed in the web app:



- 
- 
- GL hence why all the trees are back
- FO I bet that means that those things at the are suppressing something bad in those rocks
- KI Game worth picking up?

The next steps were connecting a backend service to persist data between sessions. I chose Firebase for its good real time database, because it is easy to use and has generous pricing and free tiers.

One big challenge in this process was the implementation of authentication. Firebase does have solutions for authentication in place, but authentication should only be possible using Twitch's process, which is similar to oauth2. Users are able to log-in using only their Twitch account and all the data is persisted in the real time database of Firebase. To authenticate users via Twitch, a custom authentication function was written and published using Firebase functions, which allows the deployment of server-side code encapsulated in a node.js function and executed via a range of triggers like e.g. an http get request. In the following image, you can see the popup window that opens when clicking on the "Login To Chat" button on the top right of the web app.



Once the user is authenticated, all of the data is persisted using Firebase. This data includes the user's trained Neural Network as a JSON string and the user's voted messages as an array of message objects including the message text and whether it was "liked" or not, indicating that this message should be classified as "show" or "hide".

Finally, the part of training the ANN was implemented. The [brain.js](https://brain.js.org/) library was used to train a Feedforward Neural Network. In the beginning, an arbitrary number of hidden layers was chosen after blindly testing a couple of variations. The library allows to change the number of hidden layers, as well as the learning rate.

Training was implemented by first shuffling the whole dataset of voted messages, creating the necessary data structure by creating the bag-of-words presentation for every message and finally feeding the Neural Network the training data in the needed format. The whole training process kicks-off in the *handleTrainMessagesReceived* function inside 'src/redux/actions/chat.js' and carries on in the *train* function of 'src/api/NeuralNet.js'.

---

Predicting the class “show” or “hide” of a new message was implemented by transforming the message to the bag-of-words representation and then calling the *run* function on the already trained Neural Network. This returns an array of probabilities for every class, which are only two in our case. Based on the higher probability, the message is classified either as “show” or “hide” and will be either added to the ‘messages’ state or ‘hiddenMessages’ state.

The ‘hiddenMessages’ state was added to allow the user to check if there are any false negatives and vote them up to add more cases to the dataset and essentially improve the quality of the dataset.

As a last improvement, saving of the trained Neural Network was implemented. Any trained NN of the brain.js library can be saved in JSON representation and also loaded from JSON representation. This allows for very quick loading of existing models without always retraining a model, which is a significant improvement, because training a Neural Network can take between 5 and 30+ seconds, based on the size of the dataset.

## Refinement

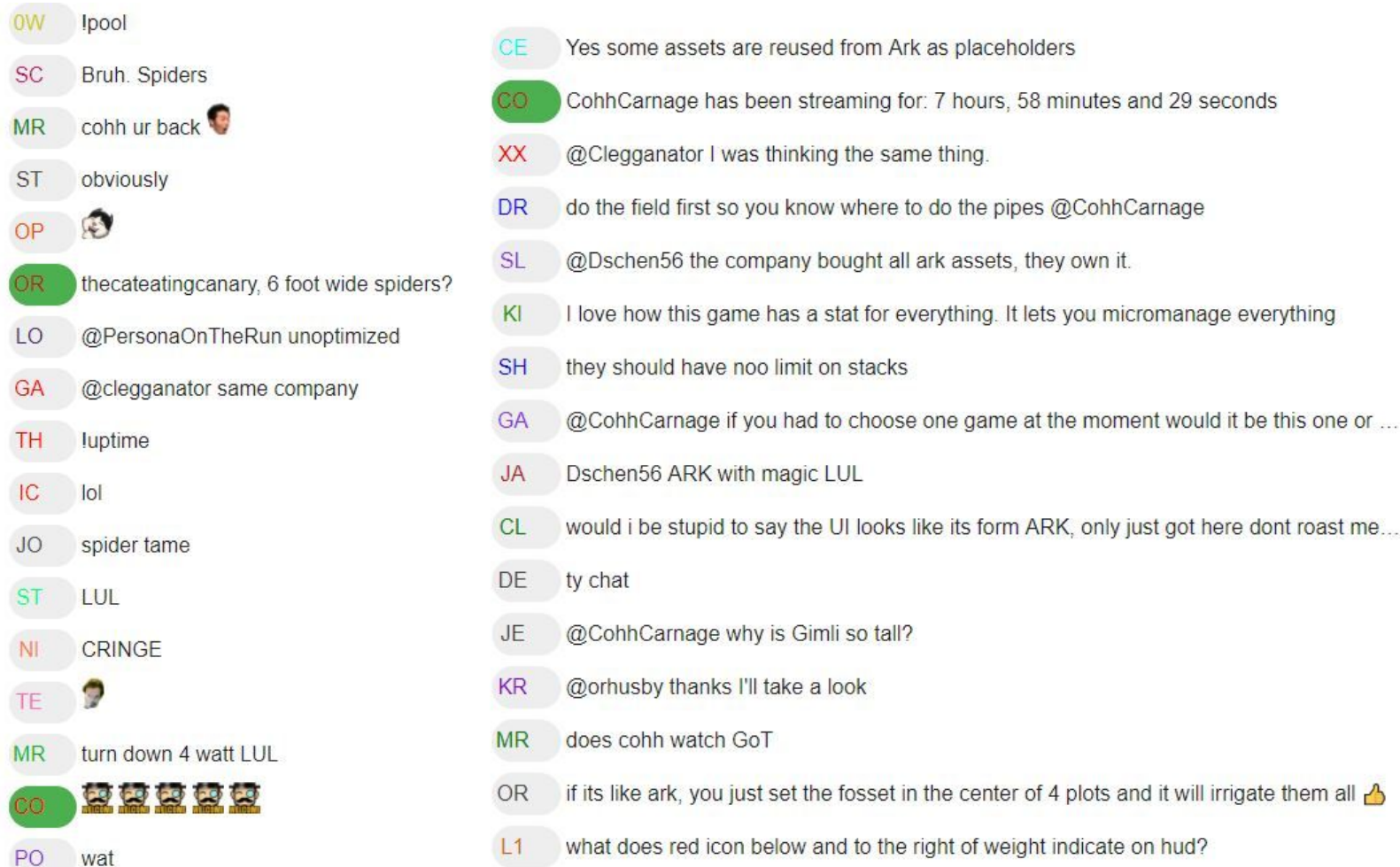
In the ‘/node’ directory you can find the file ‘hyperparamTuning.js’, in which I implemented a custom way of finding the Neural Network settings that yield the best result.

To achieve this, first all messages have been shuffled and a train test split has been created, where 80% of the data were used for training and 20% were used for testing. In the next steps, multiple variants of Neural Networks were created. Some had different hidden layer layouts and others also tweaked the learning rate.

After training every model on the training data, the test data was used to predict and outcome and calculate the model’s performance. For measuring the performance the precision, recall and F1 score have been calculated.

After multiple rounds that have been documented in the ‘node/testResults.txt’ file, the model using 1 hidden layer of 2 nodes and a learning rate of 0.05 performed best, with an average F1 score of 0.916. This is also better than the benchmark of 0.9 set for this project. Based on these results, the model of the Web App has been changed to use these best settings.

To allow the reader a subjective decision whether the problem has been solved, here is an image that shows the hidden messages on the left and the filtered chat on the right:



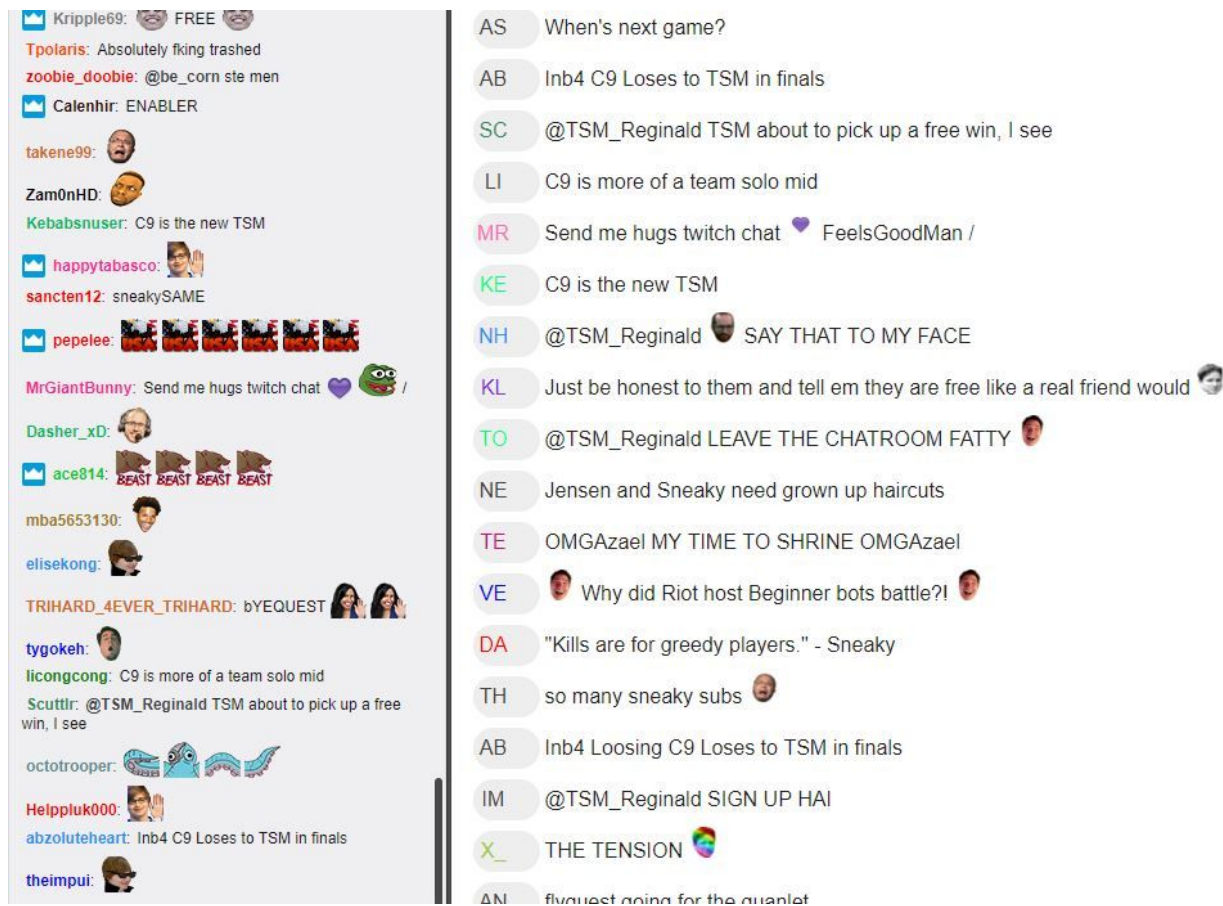
## Results

### Model Evaluation and Validation

As hyperparameter tuning was done while evaluating the best set of options for the model, we can be confident to have the best performing model possible. The final model's performance has also been stable at an F1 score of ~0.91 during multiple test runs with randomly shuffled test and training sets.

The nature of the project makes it very easy to verify the robustness, as we can easily use the trained model in production, by joining any Twitch channel we like, preferably a channel that was not used to train the model. As an example, the channel 'nalcs1' has

been used to test the robustness. Left is the original Twitch chat, right is our filtered one:



## Justification

Our benchmark of a F1 score of 0.9 has been met by the model that was found during the hyperparameter tuning process.

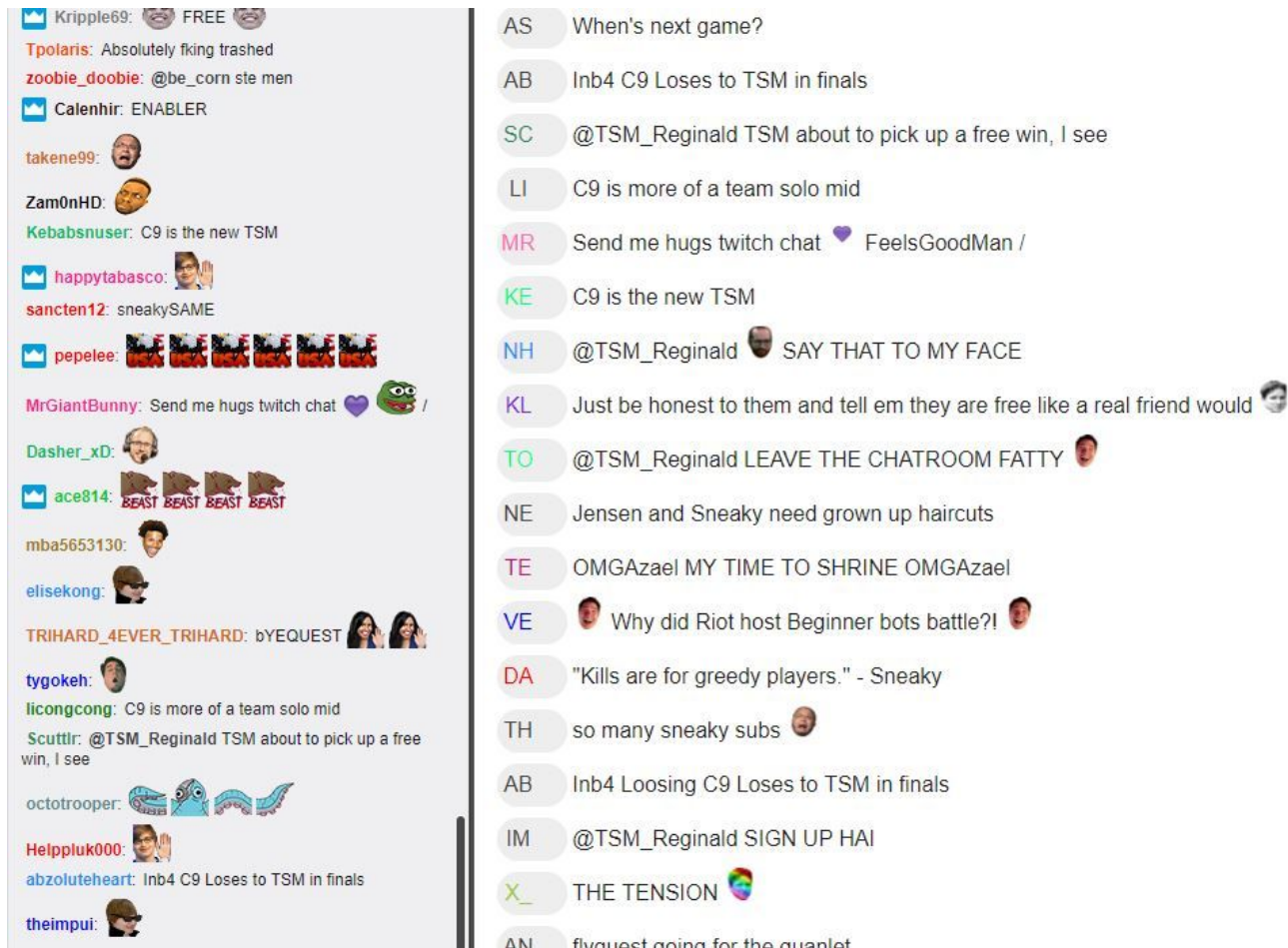
When testing the model out in the wild on any Twitch chat, it also performs very well across the board. I was originally not too confident about the performance of a Neural Network that was trained inside a Web Browser. The brain.js library has a far better performance than I thought. The ability to save and load trained models is also a very good feature of the library, as it allows to train models offline and deploy them directly with a Web App.



## Conclusion

### Free-Form Visualization

If we look again at the comparison between the original Twitch chat and this project's filtered chat, we can analyze the benefits of our better Twitch chat.



One can see that a lot of noise from random emotes and text without any value has been hidden in the project's chat on the right. E.g. the message "sneakySAME" is hidden, as are all of the messages that contain just emotes. The result is a much cleaner chat with far less messages appearing per minute and every single message has some value to it, as they are actual text messages with content, rather than unrecognizable spam.

---

## Reflection

I chose this project as I wanted to develop an application that could actually be used, instead of simply analyzing an existing dataset and trying to apply a Machine Learning model to it.

As I just started with web development this year, one main goal of mine was to really get into one front-end and one back-end framework. I chose React and Firebase and am very happy that I did so. After a lot of challenges and re-design of specific systems in the beginning, I think I am on a good path to success for this web app. The result of the Neural Network was also very satisfying. Even the first version of the model worked pretty good and had a very strong impact on the general feel of the chat, as a lot of spam and noisy messages were already hidden.

The brain.js library seems to be very powerful and is still actively maintained. I think especially the ability to save a trained model in a simple JSON representation has a huge potential. For this project, one next step could be to allow users to share their trained Neural Nets. It could be interesting to see which Neural Nets are created by users. There could be some that create models that only show very nice messages, or some that show only messages containing emotes.

Finally, the biggest issue was to come up with a project that I could manage, would like and had some kind of real world impact. I stuck some time with my initially proposed project, but was not really happy with it. After starting this better Twitch chat project, I also had to struggle with a lot of time issues, as I just started a new job and am in my wedding planning phase. Implementation wise, I think the most difficult task was to figure out a way to gather a dataset, train a neural network and use a neural network to classify right in the browser. Initially I thought about implementing the training and predicting server-side, but after I found and brain.js, there was no need for that anymore.

## Improvement

I see a lot of space for improvements, which is a good thing. The most important improvements that could be done:

A general model could be deployed that every new user loads when using the Web App. Right now, a user has to vote on messages and train their own model locally at least

---

once, before seeing a filtered chat.

Currently, there are some problems when trying to use the Web App without being logged-in using a Twitch account. It should at least be possible to use the chat and load the mentioned general model to filter messages.

Major improvements could be made to enhance the filtering. As we use a bag-of-words approach, the context between words is lost. Techniques like latent semantic indexing or using the lexical database WordNet could increase model performance. Furthermore, channel-specific text like emotes could be filtered out in an additional pre-process step to prevent any biases.

Finally, one last improvement could be made by training the Neural Network on a service worker instead of the browser's thread. This would fix the current issue, where the browser freezes for 5-30 seconds while a Neural Network is trained.