Final Report for CS 175, Winter 2021

Project Name: Chat Interpreter/TheChatInterpretersCYL

List of Team Members:

Nate Yeo, Jingtian Li, Duo Chen

1. Project Summary

Streaming is a trending online content-creating recreation/occupation where streamers interact with viewers via chat messages while live-broadcasting their desired activity (e.g. gaming). Because highlight-worthy content is only a small fraction of the stream, our project aims to use stream live chats to pre-process vods and to reduce editing time by finding potential clips. Further automation of the editing process can be done by implementing an emotion analysis model on the result.

Assuming chat increases in speed when reacting to exciting or interesting content (further development after week 7 does not have this assumption), we make clips based on chat speed, shortening the vod. We then try to classify the clips using LogReg, MLP, RNN, and GRU to filter unwanted clips.

---

2. Datasets

Our data sets were obtained via a publicly available software named TwitchDownloader (https://pypi.org/project/tcd/). This software downloads chat messages from a given stream URL and stores them into a processable json file. The json files contained much more information than we needed, such as the viewer's name, so we only extracted the timestamps and text from each chat message.

As far as we know, there are no suitable data sets online because Twitch chat evolves over time, and many of the available datasets are several years old (such as the Harvard dataset we found previously). As a result, our data is manually collected and labeled. The size is not as big as what you would see from other online data sets, but it's enough to train our learner models.

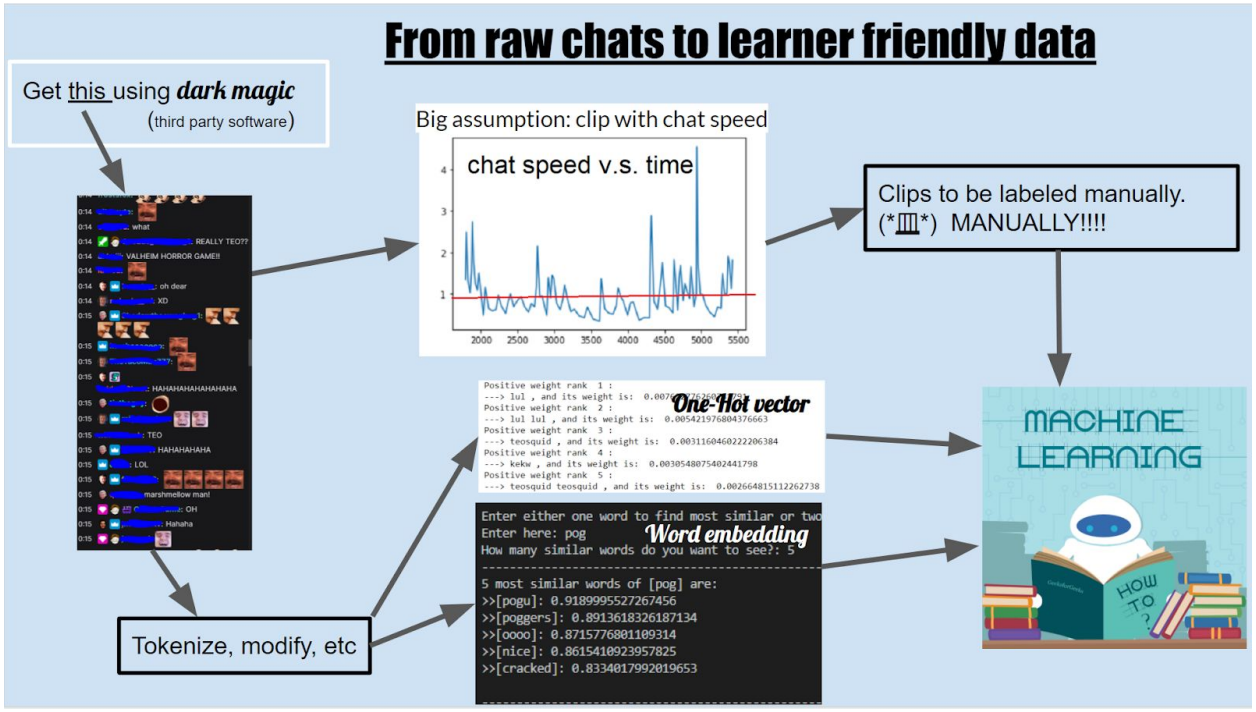
Our data is contained in a folder called "chatjsonfiles," which contains a json file for each stream, with each file containing information such as chat message and timestamp. There is also a text file in the folder called "streamURLS.txt" which has the links to all the streams we used. An important note is that it seems Twitch deletes past broadcasts after around two weeks, so viewing the actual videos is no longer possible, though we still have the data.

Data Set Statistics	Current count
Number of streams	19 streams collected from 4 different streamers
Number of clips	651 total clips, all labeled
Total number of chat messages	342845 messages
Total number of unique words (before filtering and stopwords)	115014 words
Vocabulary size after filtering and stopwords	2116 words
Largest vocabulary size (of one streamer)	1804 words

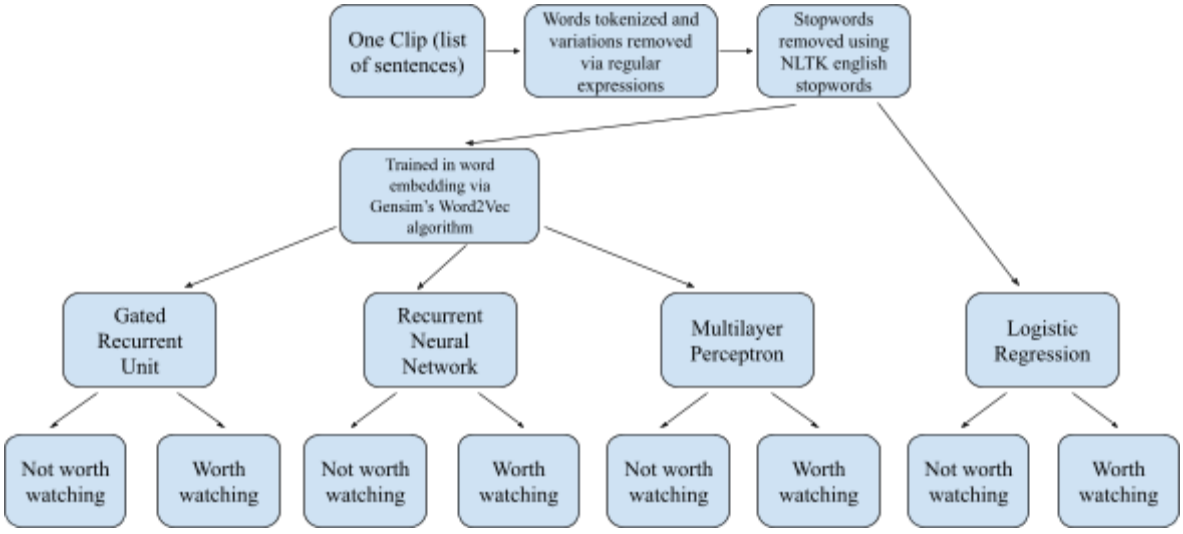
---

3. Technical Approach

Data processing flow chart:



Classification flow chart:



The pipeline each clip goes through

1) **Clipping from hours-long stream via chats:**
    - The big assumption of our project is that chat will speed up when there is good content to be clipped, but the sped-up chat does not necessarily mean good content.
    - We have a timestamp for each chat. Computing chat speed for every 30 chats minimizes noise in chat speed (Fig 1).
    - The resulting speed can be thresholded for chat intervals to crop out.

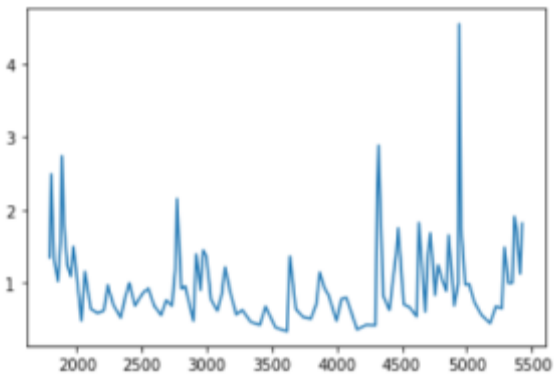


Fig 1

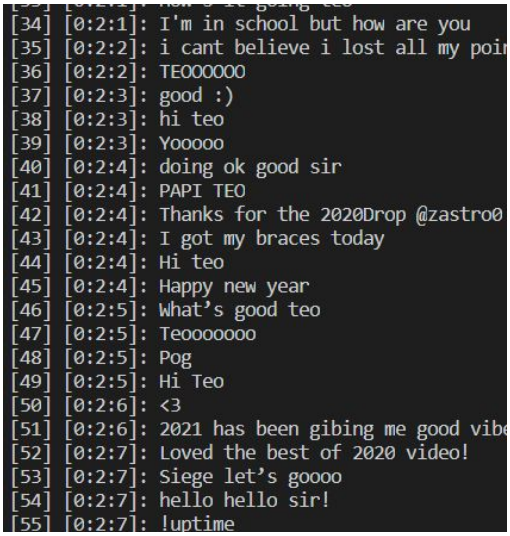


Fig 2

    - Fig 2 has chats shown in their numerical order and time stamps converted into hh:mm:ss format.
    - A clip object is implemented for program-friendly data processing and storage.

2) **Word embedding:**
    - Word embedding is one approach we have to numericalize strings. It assigns each possible word to a vector of 300 dimensions. Vector values are obtained by training a single-hidden-layer perceptron from context to word, this is done using a publicly available module named gensim.
    - To tokenize the words, the first intuition is to split by white space. However, streamers' chats are often filled with spam and special strings such as emotes, symbols, and unique slangs (such as "pogchamp," which isn't a real word). We used regular expressions and knowledge of these slangs to filter and modify the chat messages and create a token-based sentence corpus our word embedding could use.

- ➢ For example, word variations such as “niiiiicccceeeee,” and “ahahahahah…” would be considered “nice” and “haha.”

- Our embedding UI allowed us to view the information from the embeddings in several ways including printing the vectors and comparisons of the words (results of the training). In the example below, we show the 5 words most similar to ‘pog’ (a term that means nice!, or good job!, etc), and the cosine similarity between the terms ‘pog’ and ‘sadge’ (a term that is associated with sadness or disappointment).

```
Enter either one word to find most similar or two to find similarity, enter 0 to exit
Enter here: pog
How many similar words do you want to see?: 5
----------------------------------------------------------------
5 most similar words of [pog] are:
>>[pogu]: 0.9189995527267456
>>[poggers]: 0.8913618326187134
>>[oooo]: 0.8715776801109314
>>[nice]: 0.8615410923957825
>>[cracked]: 0.8334017992019653

----------------------------------------------------------------
Enter either one word to find most similar or two to find similarity, enter 0 to exit
Enter here: pog sadge
----------------------------------------------------------------
[pog]:[sadge] has similarity 0.32725024223327637
----------------------------------------------------------------
```

3) Logistic regression:

- First, it extracts all the sentences from the training data set, and tokenizes it. We designed a word purification process which transforms informally spelled words/slangs into properly spelled words. This process can also filter out special strings converted from emotes, but we found it lowered the accuracy rate. After the purification, we used nltk.tokenize.word_tokenize() to tokenize it.

- Make the bag of words. We used CountVectorizer() and TfidfVectorizer(). Users can choose between these two methods. We did not observe a significant difference in accuracy rate between them.

- The next step is to use sklearn.linear_model.LogisticRegression() to train the linear model and print out the validation AUC score and accuracy rate for result assessment.

- The classifier or model obtained from the last step should also be able to be used on other clip data. But since different chat sets may have different sets of tokens, we need to process the test data before using the model. We decided to use a big one-hot vector to include all tokens from both the training set and the testing set to solve this problem.

4) **MLP:**

- Embedding allows us to convert tokens into vectors. But each clip has arbitrary numbers of tokens, and MLP only takes fixed size input. The approach used for now is:
  - ➢ To convert chat into a bag of word dictionary
  - ➢ Sort tokens by frequencies
  - ➢ Take only the top part few tokens
  - ➢ Sum and normalize the token vectors
- The intuition is that a directed strong reaction in chat will generate repeating tokens, and those tokens likely represent the general emotion of chat. (learner output in Section 4)

5) **RNN:**

- Most of the RNN module followed assignment 2. However, we used pytorch cross entropy for binary classification, and used our customized word embedding for token vectorization.
- Because the lengths of chats can vary between 30 and 350, we decided to create a method to decrease the RNN feed numbers. The approach we used is to shift a chat window with overlappings. For example (this is implemented in Data_converter.py):
  - ➢ We have 50 chats from [0,50], chat window of 20. And overlap of 5
  - ➢ We will have windows of indexes [0,20], [15,35], [30,50], [45,50]
  - ➢ Each of the chat windows is tokenized, vectorized (using our word embeddings), summed, and normalized
  - ➢ That list is fed sequentially to the RNN.
- Because we have limited amounts of data, training takes a few hundreds epochs over them with slightly lower learning rate. Including the chat window, all our parameters are tuned through trial and error. Additionally, because of different chats' personalities (speed, language, tone, etc), each streamer's chat gets a different set of parameters. (Result of the most clipped streamer in Section 4)

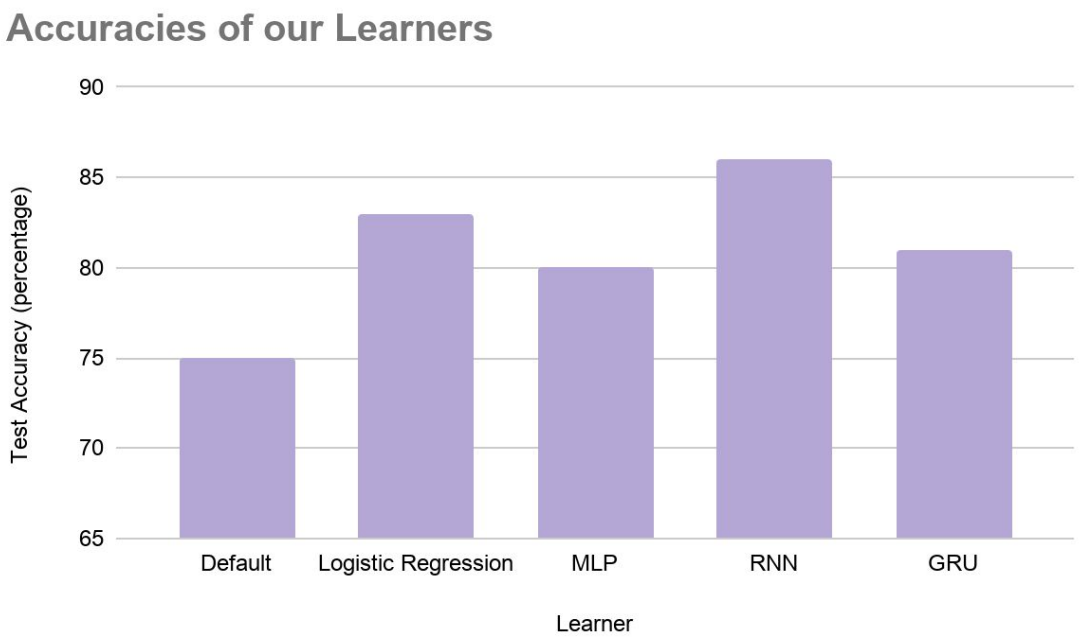
6) **GRU:**

- Because Pytorch has a builtin GRU class, the implementation was very similar to the RNN implementation. The GRU differs from the RNN in that it has two gates: the update gate and the reset gate. The reset gate determines how much of the past information to forget, while the update gate determines how much of the past information to retain. Our inputs were the same word embeddings and inputs we used in the RNN (Results in section 4).

7) **ClipperV2:**
   - The previous clipping method assumed that stream chat increases in speed when reacting to interesting content. However, we have found that this indirect approach introduces a considerable amount of noisy chats in each clip. So instead, we want an approach that orients on chats' emotions directly.
   - One intuition is that humans watch stream vods chronologically, meaning each time interval contains information with or without chats. Thus we partitioned the entire chats into equally-chronologically-intervaled sections (3 second is found to be the best length).
     - ➢ Our clipping principle is based on the homogeneity of chats within intervals. If the tokens within an interval are directed similarly (based on embedding vectors), that interval gets a mark.
     - ➢ Marking all intervals gives us a binary mask, over which we ran a blurring filter to connect small gaps between sequences of 1s.
     - ➢ Then we threshold the mask to get intervals corresponding to times when chats have homogeneous reactions. We assume those are when clips happen. (result is in Appendices)

---

4. Experiments and Evaluation



We split our data into training and testing sets (default 80/20) and the results of each learner are shown above. As you can see, there is not a significant difference in accuracies

between the default and learners. This is most likely due to the fact that our dataset was fairly small (because of how time consuming manually labeling the clips were).

Results analysis for Logistic Regression (baseline model):

The results of our logistic regression were pretty unstable. Shuffling the data every time, the majority of results were around 80-89%, and we occasionally got something really close to the default accuracy, around 77%. Since the majority of our data are labeled 0, the random shuffling can be a significant factor of the validation accuracy rate. After we found the optimal parameter for the logistic regression model, the adoption of the word purification process and tf-idf did not make a noticeable difference in accuracy. So we came to the conclusion that our log regression model had little room for improvement.

When tuning this model, we used all data. Since it can be trained relatively fast, we may train multiple times using randomly selected training and validation data. Finally we got the optimal parameters which are tolerance of 0.3, max iteration of 10, gram size of 2.

We also used other videos to evaluate this learner. We watched the clips labeled “worth watching” by our log regression model. Our assessment is that the majority(~60%) of the interesting clips proposed were fun to watch.

Tuning Hyper-parameters and results for MLP, RNN, and GRU:

Because we have a small dataset, using parameters similar to ones in the previous assignments gives us worse results than default. Hence our models all trained on very large numbers of epochs and very small learning rates to avoid side-stepping around minimas in the loss function.

The resulting testing accuracies of the multilayer perceptron were pretty inconsistent. It often depended on the training and testing data split (best one was around 83%). In addition, we found that different streamers needed greatly different parameters. For example, the highest accuracy when testing on the streamer, Teo, came with a hidden layer size of 90 while the highest accuracy for Wardell had a hidden layer size of 3. We suspect the reason is the variability of chats’ reaction. Maybe a more complex chat is better represented by a larger hidden size.

The recurrent neural network results were fairly consistent once we found the optimal hyperparameters. However, these parameters would change drastically based on the popularity of the streamer and size of their chat. We found that for large streamers such as Teo and Wardell (who often get thousands of viewers at a time), an input size of 20 chats with an overlap of 3 per input produced the best results (refer to RNN technical approach), whereas if a streamer only had around 100 viewers, these values would have to change significantly. For the streamers we tested on, however, we found that the accuracies didn’t deviate significantly upon selecting the optimal parameters (highest testing accuracy for Wardell was around 89%, while highest testing accuracy for Teo was around 87%).

The results of the gated recurrent unit actually turned out to be worse than the RNN, with the best testing accuracy being 82%. The reason for this is likely because our data is more suited to an RNN model as it isn't heavily dependent on memory retention, which is where the GRU is supposed to outshine the regular RNN.

Interpretation of result:

In addition to just printing the accuracy of the data we trained on, we manually looked at the chat messages from the mislabeled clips. What we found is that chat's increase in speed can have many unexpected reasons, such as concurrent viewer fluctuation and uninteresting stream events. Concurrent viewer fluctuation happens when the stream passes through certain hours and more people come in to watch (such as 5pm). Uninteresting stream events is a way a stream platform uses to boost chat interactions independent of stream content (such as receiving rewards for typing special messages). Additionally, many of our data involve a few hundreds of chats per clip, which introduces human reading and labeling errors. All the problems above lead to the approach mentioned in ClipperV2.

ClipperV2:

```
Clip number [33] is: [2:56:4] -> [2:57:2]
Clip number [34] is: [2:57:22] -> [2:58:20]
Clip number [35] is: [2:59:10] -> [3:0:8]
Clip number [36] is: [3:0:22] -> [3:1:20]
Clip number [37] is: [3:4:10] -> [3:5:8]
Clip number [38] is: [3:10:55] -> [3:11:56]
Clip number [39] is: [3:12:10] -> [3:13:11]
Clip number [40] is: [3:18:19] -> [3:19:17]
Clip number [41] is: [3:20:16] -> [3:21:38]
Clip number [42] is: [3:22:40] -> [3:23:38]
Clip number [43] is: [3:26:58] -> [3:27:56]
Clip number [44] is: [3:31:16] -> [3:32:14]
Clip number [45] is: [3:49:55] -> [3:51:44]
Clip number [46] is: [4:6:13] -> [4:7:11]
Clip number [47] is: [4:11:4] -> [4:12:2]
Clip number [48] is: [4:18:46] -> [4:21:29]
Clip number [49] is: [4:22:52] -> [4:23:50]
Clip number [50] is: [4:27:25] -> [4:28:32]
Clip number [51] is: [4:29:43] -> [4:30:47]
Clip number [53] is: [4:42:1] -> [4:43:35]
total time to edit is [1:0:34]
original vod duration is [4:48:42]
```

ClipperV2.py result

The image above shows the result of ClipperV2 on the stream vod https://www.twitch.tv/videos/866021945 (Twitch will delete vods that are too old for saving space, so this vod may be deleted by the time this report is read). This brings back the purpose of our project, which is to perform some automated stream processing to save editors' time. As shown in the bottom of the image, ClipperV2 is able to clip 1-hour worth of content (unlabeled) from a 5-hour-long vod. Because the result is unsupervised, we manually checked some intervals shown. They tend to properly capture chat-reactive contents, meaning that this is a step in the right direction.

---

5. Lessons Learned and Insights

For sequence-based models such as RNN and GRU, we found that they are more suitable for strictly sequential data. Take assignment 2 for example. Names are strictly sequential in a sense that each letter following a sequence is somewhat predictable. However, it is harder to predict live stream chat messages given context because a few thousands of people are typing concurrently, and chats are directly related to stream content instead of one-another. Hence, RNN and GRU may not have practical results even if we had dealt with human labeling error. In addition, MLP and LogReg are more suitable for fixed-sized input such as images, but live stream chats vary greatly in length.

Many of our available complex models require intensive training. Thus, the first approach when we use self-obtained data should have been unsupervised models such as ClipperV2 (who only detects emotional homogeneity but not classification). Moreover, when faced with a big, complex, and potentially unpredictable data behavior, we should not rely on a big and complex model to do the job, but break them into not smaller but less noisy subproblems. Again, ClipperV2 is that kind of approach where we solved the subproblem of less-noisy clipping. Additionally, that was possible because the subproblem of numericalizing diction was solved by using Word2Vec.

For log regression, even though tuning its parameter can still improve it a little, overall this model performs poorly compared to neural network based models such as the RNN. The reason could be the high number of dimensions of our data. And effort that can reduce the dimension (word purification) would probably damage the integrity of the original data.

Additionally regarding log regression, we could instead only use the most significant words to do classification, but using that technique would require a long time to pre-process the input data (find the least significant words, remove some of it; do it iteratively until some limits). And it might make the log regression model perform badly when they are trained and tested on different streamers since the most significant words are often the strings that are converted from channel-specific emotes.

Appendices

Appendix A

Publicly Available Code	Code we have written ourselves
TwitchDownloader, https://pypi.org/project/tcd/ Input: URL to a twitch video Output: a .json file contains chat messages **Imported libraries:** Nltk python module Python regular expressions Sklearn python module Gensim word embedding module Pytorch tensor and learner module	***Clip object***: self-contained object representing one clip. It stores data such as the chats of asid clip, time stamps, start/end time, and labels. ***Utilities***: a collection of commonly used methods for any module to use. Example includes: prompt user for a float (does range and error checking), prompt user for file path (also does error checking itself). ***Clipper***: generate clips from json files downloaded by TwitchDownloader and store them into specified clip files. ***Labeler***: Takes a clip file, iterates through all clips and prompts the user to manually label each clip. ***Tokenizer_kit:*** A collection of string-processing methods. ***Embedding***: Prompts the user to enter a collection of json files downloaded from TwitchDownloader, extracts chat strings, tokenizes, and trains a word embedding on said chats. Word embedding is stored in a specified file. ***Data Loader***: Collection of methods for a learner to load in data from clip files. When called as main, it inspect a specified clip file. ***Logistic Regression code***: give user prompts to train and test a log regression model ***Data_converter:*** A collection of methods for learners to call on and convert saved files into learner processable data structures. ***NN-based learners:*** Train and test a specified learner on some collection of clip files. MLP, RNN, GRU are strictly templated. They have the

	same input/output, but only differ in the model they use. ***ClipperV2:*** Process-based .py file that clips a vod that is pre-processed by data converter to be chronologically segmented. Output is printed out time-intervals specifying clips.

Appendix B

These are screenshots of the terminal outputs of running some of our files.

Clipper

```
Enter json file path (WITH .json, enter exit to exit): chatjsonfiles/TeosGame[0].json
Do you want ot use default values? (1 for yes, -1 for no, 0 for more info): 0
============================================================
value for chat window is 10
the value for min_clip_len is 10
the value for threshold is 100
============================================================
Do you want ot use default values? (1 for yes, -1 for no, 0 for more info): 1
Do you want to ignore notice chats? (y/n/i, i for more info): i
============================================================
Typically in twitch chat, every subscription is considered a chat
If someone gift subs to someone, those will maximize chat speed in that interval
you can choose to ignore those or take those into account.
Default value is True
============================================================
Do you want to ignore notice chats? (y/n/i, i for more info): y
Number of chats ignored is 355
Number of clips found is 55
How do you want to name this pickle file? (WITHOUT .pkl):
```

Output screenshot of Clipper.py

Labeler

```
Enter pickle file path (WITH .pkl, enter exit to exit): clip_data/wardell[1].pkl
============================================================
labeling 1/33 clips
video id is 882205665
chat duration is from 0:7:11 to 0:7:21
there are 40 chats
Do you want to see the chat? (y/n/e, e for exit): y
printing chat
------------------------------------------------------------
[0] [0:7:11]: !specs
[1] [0:7:11]: nice
[2] [0:7:11]: HAHAHAHAHHAHAHAA
[3] [0:7:11]: KEKW
[4] [0:7:11]: YIKES
[5] [0:7:11]: KEKW
[6] [0:7:12]: hahahahahhaha
[7] [0:7:12]: KEKW
[8] [0:7:12]: LULW
[9] [0:7:13]: loool
[10] [0:7:13]: imagine someone gift youu subs
[11] [0:7:13]: !keyboard
[12] [0:7:13]: LMAOOOOOOOOOOOOOOOOOOOO
[13] [0:7:13]: Logitech G PRO Keyboard TKL: tsm.adv.gg/a/zvahuI
[14] [0:7:13]: xqcM YOINKED
[15] [0:7:14]: lmfaoooo
[16] [0:7:14]: LEAKED
[17] [0:7:14]: WARDELL 1V1 ME PUSSY
[18] [0:7:14]: super cool alert
[19] [0:7:14]: lol
[20] [0:7:14]: steve
[21] [0:7:14]: exposed lmfaoooo
[22] [0:7:15]: !exitlag
[23] [0:7:15]: Get Rid of Lag! Try the professional players choice! Sign-up with Wardell's link to support
[24] [0:7:15]: wheeze
[25] [0:7:16]: KEKW
[26] [0:7:16]: hahahahahahaha
[27] [0:7:16]: LMFAO
[28] [0:7:16]: lmao
[29] [0:7:17]: lol
[30] [0:7:17]: KEKW
[31] [0:7:18]: how soed
[32] [0:7:18]: leakedddd
[33] [0:7:19]: WARDELL IS DONE
[34] [0:7:19]: WARDELLICIOUS!!
[35] [0:7:19]: Them notis PowerUpR
[36] [0:7:19]: Steve
[37] [0:7:19]: monkaS
[38] [0:7:20]: LMAOOO
[39] [0:7:20]: steve kekw
true label is [0]: unlabeled
true binary label is [0]

Enter a label index in int: 1
```

Example of labeling one clip using Labeler.py

Embedding

```
Enter either one word to find most similar or two to find similarity, enter 0 to exit
Enter here: pog
How many similar words do you want to see?: 5
----------------------------------------------------------
5 most similar words of [pog] are:
>>[pogu]: 0.9189995527267456
>>[poggers]: 0.8913618326187134
>>[oooo]: 0.8715776801109314
>>[nice]: 0.8615410923957825
>>[cracked]: 0.8334017992019653

----------------------------------------------------------
Enter either one word to find most similar or two to find similarity, enter 0 to exit
Enter here: pog sadge
----------------------------------------------------------
[pog]:[sadge] has similarity 0.32725024223327637
----------------------------------------------------------
```

Results of Embedding.py

RNN (as a typical output for learner)

```
================================================================
Number of clips found: [307]
Shuffling clips
Splitting train and test on ratio: [0.2]
Processing train clips
processing test clips
Training...
Default training accuracy is: [0.7580645161290323]
Training accuracy is: [0.9032258064516129]
Default test accuracy is: [0.7551020408163265]
Test accuracy is: [0.8326530612244898]
Total number of mislabeled clips is: [47]
save file? (y/n): y
file will be stored as ./mislabeled/<YOUR FILE NAME>.pkl
Enter file name (WITHOUT .pkl): Teo[1].pkl
```

Output of RNN.py

LogReg

```
enter a path to a file or a folder to add that to the training set, enter e to exitlabeled_clip_data/Teo
enter a path to a file or a folder to add that to the training set, enter e to exitlabeled_clip_data/T90
enter a path to a file or a folder to add that to the training set, enter e to exitlabeled_clip_data/viper
enter a path to a file or a folder to add that to the training set, enter e to exitlabeled_clip_data/wardell
enter a path to a file or a folder to add that to the training set, enter e to exite
What proportion of the training data would be used for validation?0.3
Do you want to use default english stopwords or stopwords given by my author? (default/author)default
Do you want to use tfidf on ohv construction? (yes/no)yes
enter y to look at top 5 significant terms, enter other to quity
Positive weight rank  1 :
---> pog , and its weight is:  2.492924087586299
Positive weight rank  2 :
---> kekw , and its weight is:  2.0105795976642042
Positive weight rank  3 :
---> lmao , and its weight is:  1.8998621100440032
Positive weight rank  4 :
---> lul , and its weight is:  1.4004352895204573
Positive weight rank  5 :
---> ??? , and its weight is:  1.2629338919091522
Negative weight rank  1 :
---> daut , and its weight is:  -1.077329887785476
Negative weight rank  2 :
---> teo , and its weight is:  -1.0091935392307345
Negative weight rank  3 :
---> teosgame , and its weight is:  -0.9000540702570717
Negative weight rank  4 :
---> daut daut , and its weight is:  -0.8939831917450652
Negative weight rank  5 :
---> teosonemore , and its weight is:  -0.8424346353847438
validation accuracy rate is -> 0.8163265306122449
default validation accuracy rate is -> 0.6989795918367347
```

Output of log_regression.py

ClipperV2

```
Clip number [33] is: [2:56:4] -> [2:57:2]
Clip number [34] is: [2:57:22] -> [2:58:20]
Clip number [35] is: [2:59:10] -> [3:0:8]
Clip number [36] is: [3:0:22] -> [3:1:20]
Clip number [37] is: [3:4:10] -> [3:5:8]
Clip number [38] is: [3:10:55] -> [3:11:56]
Clip number [39] is: [3:12:10] -> [3:13:11]
Clip number [40] is: [3:18:19] -> [3:19:17]
Clip number [41] is: [3:20:16] -> [3:21:38]
Clip number [42] is: [3:22:40] -> [3:23:38]
Clip number [43] is: [3:26:58] -> [3:27:56]
Clip number [44] is: [3:31:16] -> [3:32:14]
Clip number [45] is: [3:49:55] -> [3:51:44]
Clip number [46] is: [4:6:13] -> [4:7:11]
Clip number [47] is: [4:11:4] -> [4:12:2]
Clip number [48] is: [4:18:46] -> [4:21:29]
Clip number [49] is: [4:22:52] -> [4:23:50]
Clip number [50] is: [4:27:25] -> [4:28:32]
Clip number [51] is: [4:29:43] -> [4:30:47]
Clip number [53] is: [4:42:1] -> [4:43:35]
total time to edit is [1:0:34]
original vod duration is [4:48:42]
```

ClipperV2.py result