Apache Storm
Twitch Sentiment Analysis Pipeline

Student Name	Colin Fitzsimons
Student Number	A00045951
Github Link	https://github.com/CFitzsimons/twitch-chat-running-sentiment
Video Link	https://youtu.be/uBHqfHxCUdc

Introduction

Twitch is an online livestreaming platform[1] originally set up with the focus of allowing users to livestream themselves playing video games. It has since evolved to become a platform not just for video games, but also for general livestreaming use cases. As of the date of this publication, the average concurrent viewers is in the ballpark of 2.18 million[2]. Each livestream has a chat where viewers can interact with each other and the host. Although Twitch themselves do not release the number of messages being sent at any given time, it's not unreasonable to consider it a large number.

The volume of messages poses a genuine problem for Twitch from an infrastructure point of view, but it also raises some concerns for the livestream hosts themselves. It has been noted in other literature that audience management is both difficult[3], and necessary. If hosts and moderators are not able to keep up with the messages, the chat can quickly devolve into a derogatory, negative space[4].

Prior research has explored performing sentiment analysis on a gathered set of chat messages, often quite a long time after the livestream itself has ended[5], [6]. This is too disconnected from the actual livestream to be useful to moderators, or the livestream host themselves.

This paper introduces an Apache Storm[7] pipeline that aims to perform sentiment analysis in realtime, during a livestream session. The system aims to provide the tools necessary for moderators and hosts to curate constructive chat environments during their streams. It also seeks to offer a method for researchers to customise the analysis portion of the pipeline to create different datasets without needing to construct a full ETL themselves.

Apache Storm was selected as it is a higher level distributed framework suited for taking a set of streams and distributing their processing over many workers. The final destination in the proposed pipeline is a PostgreSQL database since it is an ACID compliant database that also supports NoSQL data through the jsonb datatype. Apache Storm consists of spouts, bolts and streams of tuples. This portion of the system was written in the native language supported for Storm, Java. However, the sentiment analysis portion was constructed as a RESTful python server, this was done so other researchers could easily swap the analysis model independently of the compiled Storm topology.

Related Work

There are a good number of projects, particularly master's thesis, that have attempted to perform sentiment analysis on Twitch chat messages. There are also a number of projects that attempt live sentiment analysis using Kafka, or similar technologies. However, this

paper and pipeline are positioned in a notable gap discovered while reading the related work. The pipeline proposed in this paper attempts to provide an extensible realtime pipeline that future researchers could adapt to trial different sentiment analysis tools. It is also positioned to allow Twitch hosts and application developers to build realtime moderation applications on top of, something the other systems explored do not provide. The following three works highlight some of the existing research and development done in this area.

Live Sentiment Analysis using Apache Flink

The Flink implementation is very powerful, but tightly couples the sentiment analysis system to the implementation[8]. This is not necessarily a downside, just a difference in approach. Implementing the sentiment analysis directly within a bolt in Storm would mean that an external REST service would not become a bottleneck, which is a potential issue with the proposed architecture implemented as part of this paper that will be discussed later. The other difference is in the connection logic to Twitch, although somewhat minor, the Flink implementation relies on a managed library called Twitch4J, which is perfectly acceptable but the Storm implementation treats the Twitch connection as simple Internet Relay Chat (IRC).

Thesis: Emote-Controlled, Obtaining Viewer Feedback through Emote-Based Sentiment Analysis

This thesis[9] focused on processing a large amount of Twitch messages after the streams had ended. The research had a focus on post-processing of the data, and was a modeling and data collection task rather than system design and development. The sentiment analysis model used was trained specifically for the processing of Twitch chat messages, focusing heavily on the use of emotes which are a common way for viewers to express emotions in a channel. This paper's pipeline does some minor preprocessing to encode emotes, but it is not the focus of the preprocessing step.

Sentiment Analysis: The Case of Twitch Chat

This paper focused primarily on rule based sentiment analysis, and again is primarily a post-processing exercise. The approaches and difficulties discussed have a lot of merit and help identify some of the problems faced in running typically sentiment analysis models on Twitch chat messages.

Dataset

This paper identifies two datasets, an inbound datasets, which are the messages coming from Twitch, and an outbound dataset. The outbound dataset is a set of features that are

added to a PostgreSQL database. These datasets are not static and as such, a direct link cannot be provided but data sources are available via the Twitch website[1].

Inbound Dataset

Since the dataset used for this paper is not static, it is live streamed into the Storm system. It consists of a constant flow of messages from various Twitch channels. Each message has the following properties:

- A timestamp
- The user the message is associated with
- The chat message
- The channel the message was sent to

The incoming messages are read as IRC messages by a Storm spout and streamed into the various data processing and analytics bolts.

Outbound Dataset

The outbound dataset consists of the following features:

- A timestamp
- The original message
- The processed message which translates emotes, and removes non-standard characters
- The channel the message is associated to
- The top sentiment
- The top sentiment weight
- A json object with all sentiment weights

The JSON object produced from the model used for sentiment analysis contains a map of a sentiment label to a weight.

This might take the following form:

```
{
  “anger”: 0.23,
  “neutral”: 0.27,
  “joy”: 0.5
}
```

Data Processing

Preprocessing Bolt

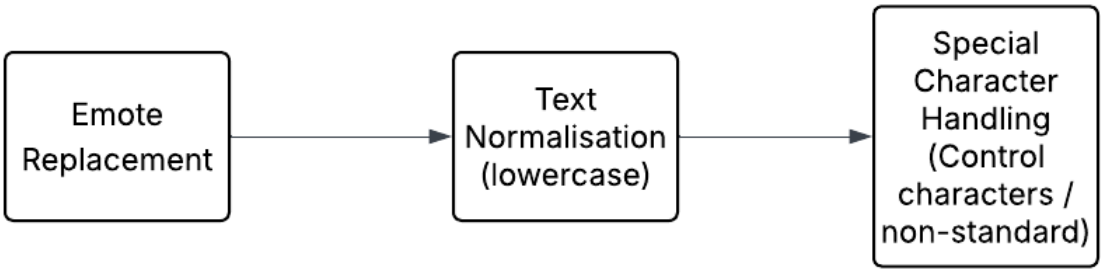

figure 1.

The preprocessing bolt has three distinct tasks as detailed in figure 1. Each serves a specific purpose and helps prepare the chat message for sentiment analysis.

Emote Replacement

This stage encodes terms such as “PogChamp” or “LUL”, which are phrases that commonly occur within a Twitch chat session, as simple phrases. “LUL” is replaced with the word “funny”, while “PogChamp” is replaced with the phrase “I am amused”. This is not a sophisticated replacement system and has obvious problems noted by one of the earlier research papers. Slang typically changes meaning over time, and these emotes are no different. A static translation is unlikely to be as relevant in months to come. However, this offers an opportunity in that it allows the messages to be encoded with the current contextual meaning of the emote. If this processing was done months later, the meaning might have shifted enough to skew the sentiment analysis.

Text Normalisation

The text is then normalised by the built-in Java Normalisation class, specifically using the NFKC form. This will decompose some special characters into their components as well as change certain text into a compatible form. Full width characters “ A B C “ might be encoded as “ABC” since the space around the characters is not actually a whitespace character.

Special Character Handling

During testing a number of special characters caused downstream problems in the sentiment analyser. These were detected and added to the special character list, they include control characters and other non-standard whitespace characters.

Sentiment Analysis Bolt

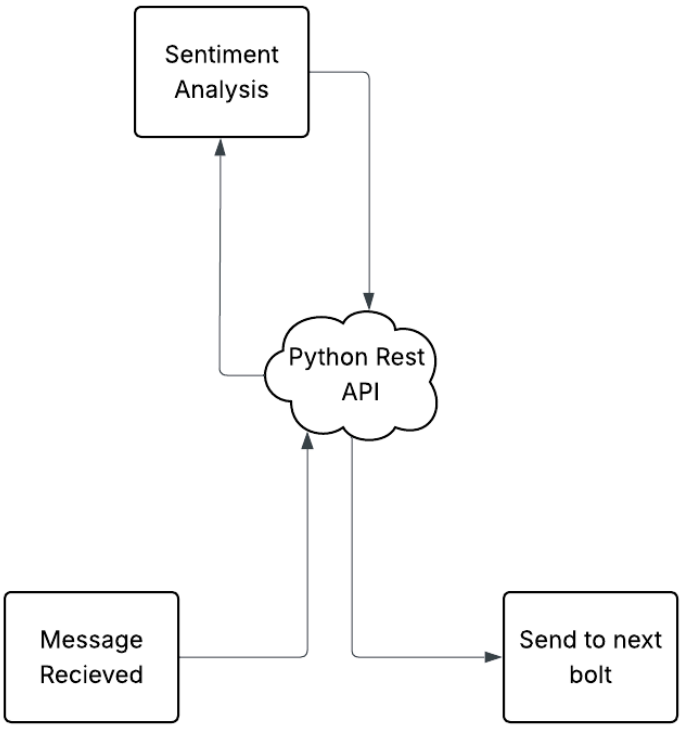

figure 2.

The sentiment analysis bolt consists of a simple HTTP call to a locally networked REST service that hosts the sentiment analysis. The message is received from the preprocessing bolt and sent to the REST api which returns a JSON payload with a set of weights. These weights are parsed and the top sentiment is identified as well as the weight associated with it. All this data is passed to the next bolt for further processing.

In the example provided, the sentiment analysis model used is a multi-label sentiment analyser[10] available through HuggingFace. This provides a modern approach to sentiment analysis using a transformer architecture. This particular model was chosen because it was fine tuned on Reddit messages, which often contain slang and are more likely to be able to accurately describe a Twitch chat message which often uses similar slang.

Pipeline Development

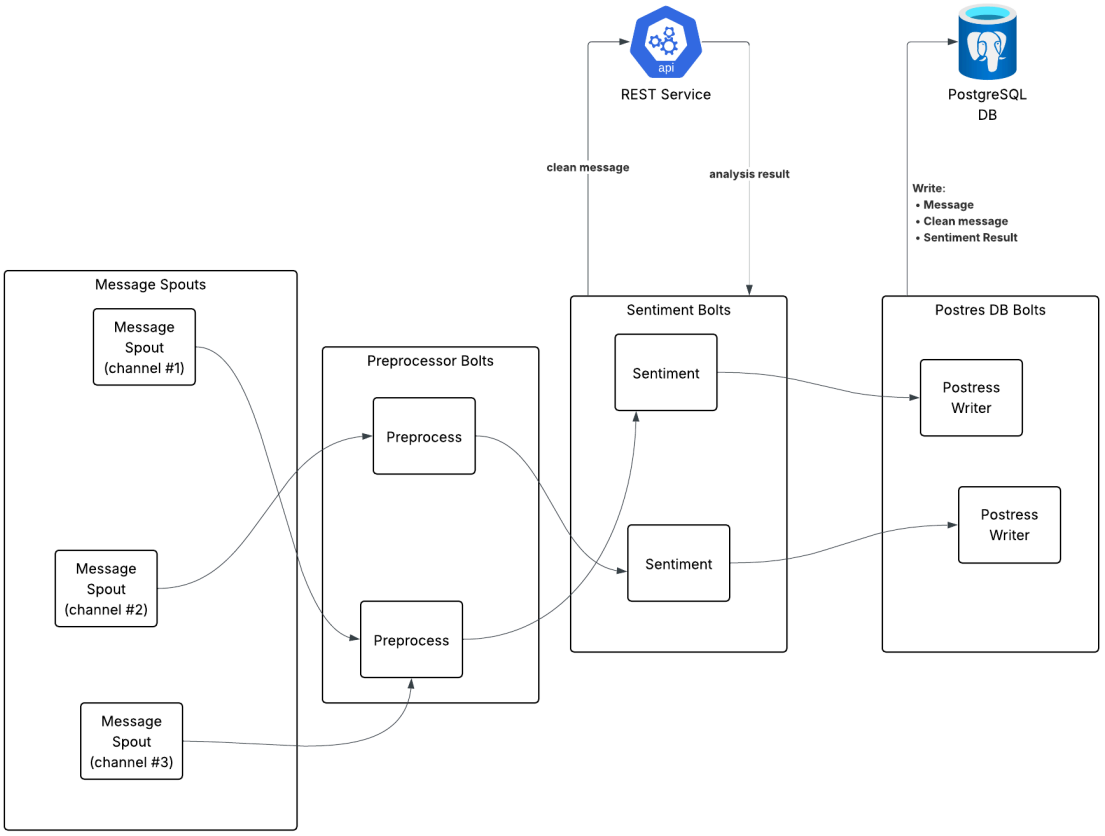

figure 3.

The pipeline developed is represented in figure 3 and consists of a set of bolts, spouts, streams and external services. The preprocessor bolt and sentiment bolts have already been discussed, but the sentiment bolt has some unique limitations that require additional discussion.

Message Spouts

The message spouts connect one-to-one to a Twitch channel's IRC chat. In the original design each Spout could handle multiple Twitch streams, which proved to be fine when subscribing to smaller Twitch channels, but quickly became unmanageable when connecting to others where the volume of messages were much higher. It's for this reason that each Spout connects to a singular channel.

The Spout establishes a connection when starting and maintains that connection until the Storm topology is killed. The tuple sent to the first bolt consists of the features listed above in the inbound data section of this paper and does not attempt to do any processing except for removing standard IRC message decorations that are noted in the IRC RFC[11].

Sentiment Bolt

The sentiment bolt is effectively a simple HTTP passthrough to a python server that actually performs the analysis. This was a decision made to support extensibility in the future but has a major drawback in the current implementation in that it is a bottleneck. The REST service is not deployed scalably, instead, it is a simple docker instance running a minimal python image. This was not done without thought, the eventual goal of this pipeline is to have this service deployed in a kubernetes cluster which would allow a future implementor to easily scale the number of analysis nodes as needed. If deployed on kubernetes it could also be hosted on specific hardware such as GPUs which might be better suited for Machine Learning (ML) tasks.

PostgreSQL DB Bolt

This is the final destination in the proposed system. Each message, now with a set of sentiment features associated, is inserted as a row into a database. This is the second potential bottleneck of the current implementation. As it currently stands, even though the bolts that acquire the connection to the external database are horizontally scalable, the database as it is deployed is not. This is a somewhat tougher problem to solve since one of the major benefits of the current database choice is that it is realtime, allowing application developers to build on top of the database without having to worry about how the data is being input.

The first way to alleviate this would be to use a managed service such as AWS's Aurora[12] which offers horizontal write scaleability. This is an expensive and proprietary solution though. The other option is to use a distributed storage system such as Apache Cassandara[13] which does support horizontal scalability. In contrast, tools such as native Hadoop would not be good candidates as they are not positioned to support low-latency, row level inserts, in the way that an ACID compliant database might.

Docker

The topology built can be deployed to any Apache Storm system with minimal changes, but the example implementation leverages docker, and a docker compose file to handle the various dependencies that are required to run this on a local system. The docker compose file handles deployment of all the services in figure 4 which highlights some of the dependencies expressed in the configuration files.

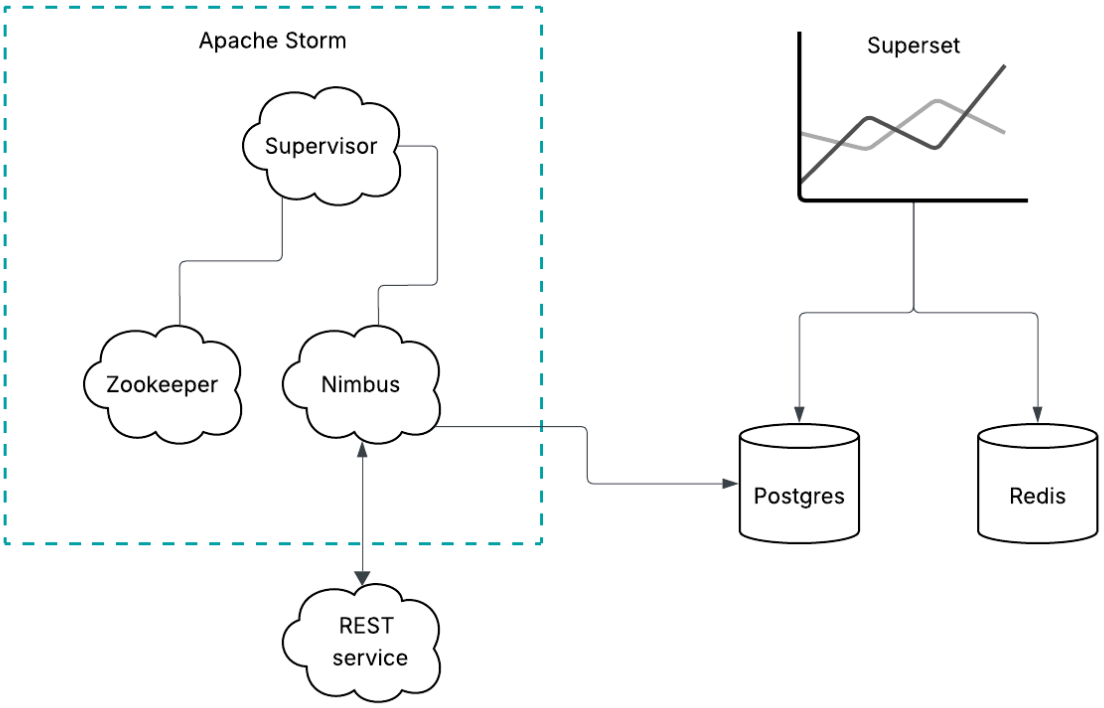

figure 4.

REST Service

The REST service is a modified python dockerfile that contains an installation of pytorch so it can run a transformer based model in CPU mode. The service was originally built using flask but was swapped to gunicorn so it could be deployed and managed in a multi-threading mode. In either case, these are python libraries that make hosting an API simple.

The external rest service is a bottleneck in the current iteration. It is run outside of the Apache Storm pipeline and is hosted in a static container. This results in the maximum throughput of the system being dependent on how efficiently the service can run sentiment analysis on the incoming messages.

Superset Dashboard

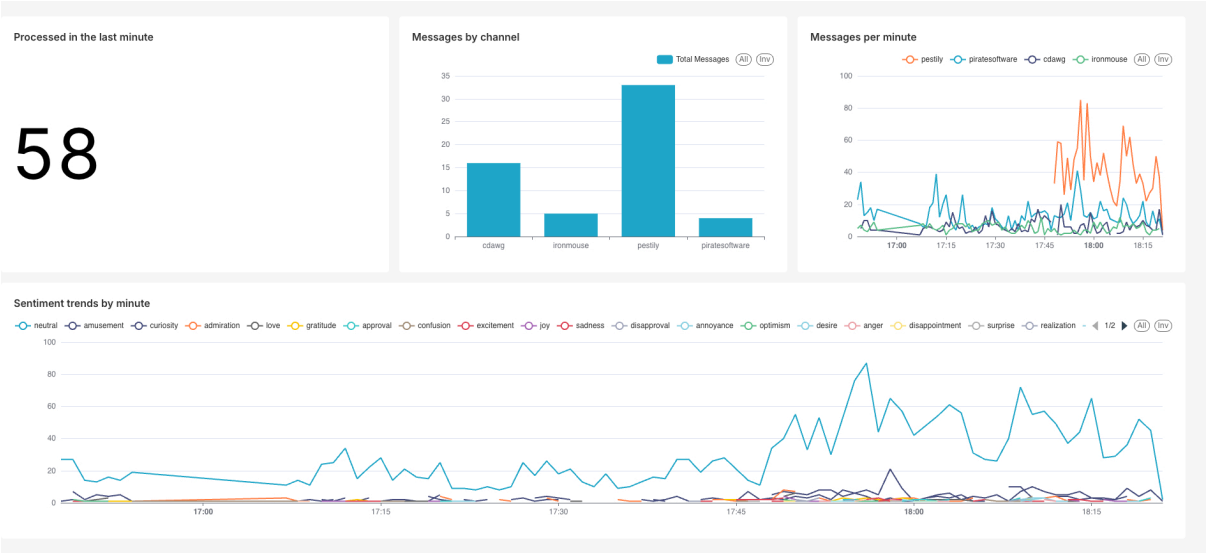

figure 5.

Apache Superset is an open source business intelligence platform that can connect to different data sources and produce charts and dashboards. This system uses it to provide an example of the realtime analytics that are possible with the pipeline. Figure 5. shows some examples of minute-by-minute analytics running against four different Twitch live streams. The top left number shows the number of messages processed in the last minute, while the lower chart shows the sentiment associated with the messages plotted over time.

Postgres/Redis

The Redis and Postgres containers are largely unchanged bar some minor port and network configurations. The Redis instance is used by Superset for query caching, while the Postgres database is used by both the Postgres bolt and Superset for persistent storage.

Challenges and lessons learned

The biggest challenge to this project was the amount of moving parts. A lot of time was spent configuring and tweaking different parts of the pipeline to interact with each other. There were also some concessions made in the interest of time such as not deploying the flask service in a kubernetes cluster. There was also a great deal of challenge in re-learning Java so a custom Storm topology could be built.

In terms of lessons learned, this has proved a valuable exercise in container orchestration, Apache Storm, and in general, distributed computing. Identifying bottlenecks and proposing mitigation strategies was an interesting challenge that helped this researcher explore tools and technologies such as Cassandra that are specifically made to solve some of the challenges faced.

It's also worth noting that independently these technologies are not difficult to implement or build upon, but creating a full, working, pipeline is more of a challenge than expected. If the task were to be repeated, it would be the recommendation of this researcher to simplify the problem and only tackle a portion of it.

Bibliography

[1] "Twitch," Twitch. Accessed: Nov. 26, 2025. [Online]. Available: https://www.twitch.tv

[2] "Twitch Statistics & Charts," TwitchTracker. Accessed: Nov. 26, 2025. [Online]. Available: https:///twitchtracker.com/statistics

[3] D. Y. Wohn and G. Freeman, "Audience Management Practices of Live Streamers on Twitch," in *Proceedings of the 2020 ACM International Conference on Interactive Media Experiences*, in IMX '20. New York, NY, USA: Association for Computing Machinery, June 2020, pp. 106–116. doi: 10.1145/3391614.3393653.

[4] T. Mihailova, "Navigating ambiguous negativity: A case study of Twitch.tv live chats," *New Media Soc.*, vol. 24, no. 8, pp. 1830–1851, Aug. 2022, doi: 10.1177/1461444820978999.

[5] K. Kobs *et al.*, "Emote-Controlled: Obtaining Implicit Viewer Feedback Through Emote-Based Sentiment Analysis on Comments of Popular Twitch.tv Channels," *Trans Soc Comput*, vol. 3, no. 2, p. 7:1-7:34, Apr. 2020, doi: 10.1145/3365523.

[6] J. Kim, D. Y. Wohn, and M. Cha, "Understanding and identifying the use of emotes in toxic chat on Twitch," *Online Soc. Netw. Media*, vol. 27, p. 100180, Jan. 2022, doi: 10.1016/j.osnem.2021.100180.

[7] "Apache Storm." Accessed: Nov. 26, 2025. [Online]. Available: https://storm.apache.org/

[8] V. Janz, "Real-time Twitch chat sentiment analysis with Apache Flink," TDS Archive. Accessed: Nov. 26, 2025. [Online]. Available: https://medium.com/data-science/real-time-twitch-chat-sentiment-analysis-with-apache-flink-e165ac1a8dcf

[9] "(PDF) Emote-Controlled: Obtaining Implicit Viewer Feedback Through Emote-Based Sentiment Analysis on Comments of Popular Twitch.tv Channels," *ResearchGate*, Aug. 2025, doi: 10.1145/3365523.

[10] "SamLowe/roberta-base-go_emotions · Hugging Face." Accessed: Nov. 26, 2025. [Online]. Available: https://huggingface.co/SamLowe/roberta-base-go_emotions

[11] "Internet Relay Chat Protocol," Internet Engineering Task Force, Request for Comments RFC 1459, May 1993. doi: 10.17487/RFC1459.

[12] "OLTP Database, MySQL And PostgreSQL Managed Database - Amazon Aurora - AWS," Amazon Web Services, Inc. Accessed: Nov. 26, 2025. [Online]. Available: https://aws.amazon.com/rds/aurora/

[13] "Apache Cassandra | Apache Cassandra Documentation," Apache Cassandra. Accessed: Nov. 26, 2025. [Online]. Available: https://cassandra.apache.org/