DATA201 Report

Group members: *Brett Ellis, Connor Smith, Jamie Hutchings, Liam Pol*

Choice of Data

At the start of the project, we were unsure about what our topic was going to be. The first thing we did as a group was discuss a few topics we might find interesting. We looked at potential applications in climate, native bird population, electricity grid data, E-commerce, and Spotify or music playlist data.

However, we ended up getting sidetracked and talking about a recent clip from a Twitch.tv livestream. We started then talking about collecting livestream data from various streaming platforms such as Twitch, YouTube, and Mixer as this was something we all had a keen interest in.

Ultimately, we decided to only use Twitch, since we thought that would be enough data to work with. In the end we were right, collecting about 530,000 messages and about 180 API pings.

Collection of Data

To collect the data, we ran 3 scripts:

- bot.js
- get_emotes.py
- import_requests.py

The bot.js script is based on a template from the Twitch documentation and uses the Twitch tmi.js node package. We modified the script to connect an anonymous client to all of the streams and streamed their irc data straight into a csv named chat.csv. This included all of the messages from every single streamer.

The get_emotes.py script used the Twitch API to get the streamer ID's for our given streamers. From there, we used these streamer ID's to then request the emotes from TwitchEmotes.com and read them into twitchEmotes.csv and channelEmotes.csv.

The import_requests.py script used the Twitch API as well, getting the streamer metadata for each of the streamer ID's. This data was then saved in individual csv files. For example, streamer metadata for Shroud would be saved in shroud.csv. However, in the end we only had 11 streamer metadata csv files, since 14 out of the 25 chosen streamers did not stream during the 24 hour collection period.

Ethics and Legality

We collected our data from four different sources:

- Twitch API

- Twitch Emotes API
- Twitch IRC Chat
- FrankerFaceZ Open Third Party Emote Library

To collect the Twitch API data we followed all of the correct procedures and registered our app with Twitch and used our client id and client secret when accessing the API. Furthermore we abided by their terms of service by not using the data for commercial purposes and not attempting to collect any data with the intent of collecting specific user data. This applies to the Twitch Emotes API as well.

Secondly, we collected the Twitch IRC data which is publicly accessible and uses anonymous aliases as usernames. Due to the data being publicly available and the anonymity there were no issues collecting this information.

Finally the third party emote library FrankerFaceZ. This data source has an API that allowed us to collect the data, however due to being a small site and lack of API documentation it was a lot easier to just use web scraping techniques. We assume the site has no problem with this as all the information is included in their open API anyway.

Overall we collected no data on specific users and there was no intent made to extrapolate sensitive data, nor do we provide any data that could be used to identify users. Therefore we conclude that not only is our data legal but ethical.

Why Use an ER Diagram

The ER diagram seemed to be the only real option to describe our data. As the data was collected, it already had the foreign keys in the table to represent the relationships. We also needed some way to concisely describe our data that did not lose any information. This then gave us an algorithm to turn our data into a relational database with little effort.

How We Cleaned the Data

The API streamer data or “metadata” was cleaned via individual functions that were then combined into one main function that processed all the csvs. All of this code is under the API Meta Data Cleaning section of the Processing notebook. Each of the functions would do a different cleaning element such as adding sensible headers, editing rows so that they were more usable for analysis, removing rows that were redundant or not useful and changing the class of columns to more practical types such as converting date_started to a date class. We then used a function that contained a for loop in order to read in the csv files, this function would return the csv if it was readable, if it would give an error the function would bypass this and just return the name of the file. All of these functions were then put in one main function containing all of the previous functions that would take a csv, read it, clean it and return it. If the csv passed in was unreadable, for example if it was empty, it would append the filename to a list containing all the other filenames that were unreadable. The next step in the code takes a folder full of csv files, puts them into a list as strings and then runs this list through a for loop containing the main function and appends all the readable csv files to one data

frame which is written back to a csv called API_PING.csv and this is out final data format of the metadata.

The chat data did not have much that needed to be done. The channel strings had one hash at the start that needed to be removed, as it cluttered up the data and prevented it from relating to the channel dataframe. This was easily done with the ‘sub’ function in R. An observation column also had to be added to form a unique key within the dataframe. This was done by using the mutate function and adding a sequence from 1 to the length of the data.

Challenges

One problem that we faced was deciding how we were gonna clean the data, what data we were going to keep/drop as well as the final format of the data. In the end we chose to just make the variables that were already there more usable such as converting some strings to numerics or dates. We also decided to not add many variables as whoever was using our data could simply operate on the columns already there to make their own. When choosing what columns to remove we decided to remove the ones we felt were not useful, such as stream title or the start time after we separated into the date started and the time of day started.

We also struggled to come to a decision on what graphs were going to make for analysis and include in our final presentation as some of the graphs we tried to make used extra variables that were created from the main data frame which gave us some issues. In the end we decided to have a more simple approach as the main focus of the project wasn't on analysis and just mention that a more in depth analysis could be done with our data.

Given that we had a large amount of data, and potential users could collect even more, we had to make sure that the ‘contains’ relation table did not take too long to be generated. Initially we tried using nested recursive functions, although, since R is not optimised for these, it would have taken multiple days to terminate on the data we had. We then tried for loops as these are much faster than recursion. However, R is still not optimised for loops and this would have taken hours to complete. We then researched how for loops should be implemented in R and found the lapply function. Using nested lapply functions, the table took a few minutes to be created. For scalability the time complexity is O(ME) where M is the number of messages, and E is the number of emotes.

We also had a few minor issues surrounding the collection of data. When running the JavaScript chat bot, we encountered issues with connectivity. In the first trial of running the data, we had 2 timeouts. To combat this, we needed to make sure that the computer running the scripts doesn’t lose connectivity, or automatically update. To combat this, we just ran another python script that simulates the pressing of a key to make sure the state of the computer doesn’t change and interrupt the programs.

At the same time, we also had issues running the bot.js and the import_requests.py concurrently. As mentioned before, it was quite prone to interruptions. We just decided in the end that we would isolate a computer to run both scripts.

Since all of us were working on different devices and on different code, it did not all work as a unit when it was combined. To fix this we went through the data, slowly debugging it. Most commonly we had to change filenames and directories, and the names of headers, since the data was not initially processed with a relational model in mind. During this stage we also made the code more readable for other users.

What would we do if we carried on with this data?

If we were to carry on with this project we would look to implement this as a whole R package. To do this we would first convert the python code to R. We would also research ways of making our data behave more like relational models. This would involve looking at packages or creating our own functions to handle data manipulation, and also potentially changing the saved files from .csv to .sqlite.