CS4624
Multimedia, Hypertext, and Information Access

Language and Sentiment Analysis of Extremist Behavior in Online Game Communities

Team: Liam McBride, Daniel Lanigan, Renzo Neps
Client: Dr. James D. Ivory
Instructor: Dr. Edward A. Fox
Virginia Tech, Blacksburg, VA 24061
5/9/2024
Table of Contents

1.0 Table of Figures ........................................................................................................... 3
2.0 Table of Tables ............................................................................................................ 4
2.0 Abstract ..................................................................................................................... 5
3.0 Introduction ................................................................................................................ 6
  3.1 Problem ..................................................................................................................... 6
  3.2 Client and Motivation ............................................................................................... 6
  3.3 General Approach ..................................................................................................... 6
4.0 Requirements .............................................................................................................. 8
  4.1 War Thunder Forums Scraper .................................................................................. 8
  4.2 YouTube Scraper ..................................................................................................... 8
  4.3 Data Collected ......................................................................................................... 9
  4.4 Sentiment Analysis Results .................................................................................... 9
  4.5 Visualizations of Data ............................................................................................. 9
5.0 Design ........................................................................................................................ 11
6.0 Implementation ......................................................................................................... 12
  6.1 Scraping .................................................................................................................. 12
  6.2 Cleaning .................................................................................................................. 13
  6.3 Visualizations ......................................................................................................... 13
7.0 Testing/Evaluation/Assessment ............................................................................... 20
  7.1 Testing .................................................................................................................... 20
  7.2 Evaluation ............................................................................................................... 21
  7.3 Assessment ............................................................................................................. 24
8.0 Users' Manual .......................................................................................................... 26
  8.1 Forum Manual ......................................................................................................... 26
    8.1.1 Setup ................................................................................................................ 26
    8.1.2 Scrape Categories and Threads ....................................................................... 27
    8.1.3 Randomize Target Threads ............................................................................. 27
    8.1.4 Scrape Posts ..................................................................................................... 28
  8.2 YouTube Manual ..................................................................................................... 30
9.0 Developer's Manual .................................................................................................. 35
  9.1 Inventory of All Files .............................................................................................. 36
  9.2 Overview: Data Shape and DB .............................................................................. 38
  9.4 Overview: Forum Scraping Process ....................................................................... 40
  9.5 Overview: Database.py ......................................................................................... 43
  9.6 Overview: YouTube Scripts ................................................................................... 45
  9.7 Future Work ............................................................................................................ 47
10.0 Lessons Learned ..................................................................................................... 48
  10.1 Timeline ................................................................................................................ 48
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.2 Problems and Solutions</td>
<td>49</td>
</tr>
<tr>
<td>10.4 Future Work</td>
<td>51</td>
</tr>
<tr>
<td><strong>11.0 Acknowledgements</strong></td>
<td>52</td>
</tr>
<tr>
<td><strong>12.0 References</strong></td>
<td>53</td>
</tr>
</tbody>
</table>
1.0 Table of Figures

Figure 1: Total YouTube Comments vs. YouTube Comments Matching Dictionary.........................14
Figure 2: Total Forum Comments vs. Forum Comments Matching Dictionary..............................15
Figure 3: Forum Comments Matching Dictionary Over Time...........................................................16
Figure 4: YouTube Comments Matching Dictionary Over Time....................................................17
Figure 5: War Thunder Forum Word Cloud..................................................................................17
Figure 6: YouTube Word Cloud..................................................................................................18
Figure 7: Number of Forum Posts on the Forum on a Leak Day....................................................19
Figure 8: Example of War Thunder Post Python Object................................................................20
Figure 9: Screenshot of Executing Test Forum Scraper in Terminal..............................................21
Figure 10: Screenshot of the Scraper Script Executing.................................................................28
Figure 11: Query output 1............................................................................................................29
Figure 12: Query output 2............................................................................................................29
Figure 13: Query output 3............................................................................................................30
Figure 14: To enter the Google application script.........................................................................30
Figure 15: Screenshot of the video scraper..................................................................................31
Figure 16: Screenshot of the comment scraper...........................................................................32
Figure 17: Screenshot of the exporter scraper.............................................................................33
Figure 18: Screenshot of the youtube-db-importer.py.................................................................34
Figure 19: Overview Image of Forums Codebase.........................................................................35
Figure 20: Overview Image of YouTube Codebase......................................................................36
Figure 21: Kickoff Python Code..................................................................................................41
Figure 22: scrape_thread Python Function..................................................................................41
Figure 23: scrape_one_slice_at_index Python Function.............................................................41
Figure 24: get_html_forum_thread_at_index Python Function....................................................42
Figure 25: make_lxml_tree Python Function..............................................................................42
Figure 26: get_post_trees Python Function................................................................................42
Figure 27: parse_post_id_from_post Python Function...............................................................43
Figure 28: store_posts Python Function....................................................................................43
2.0 Table of Tables

Table 1: Scraped Comments Goal vs. Actual................................................................. 9
Table 2: Execution times of scripts............................................................................ 22
Table 3: Metric Data..................................................................................................... 24
Table 4: Table of all files.......................................................................................... 38
Table 5: Forum Database Tables.................................................................................. 39
Table 6: YouTube Database Tables............................................................................. 40
Table 7: Forum Scraping Process................................................................................ 43
Table 8: Database.py functions................................................................................. 45
Table 9: YouTube Scraping Process............................................................................ 47
Table 10: Timeline/Milestones................................................................................... 49
Table 11: Problems and Solutions.............................................................................. 51
2.0 Abstract

Language and Sentiment Analysis of Extremist Behavior in Online Game Communities was a Multimedia, Hypertext, and Information Access capstone project to assist the VT Gamer Lab in gathering more data to analyze links between online video game communities and extremist behavior. Specifically, military simulation games (referred to as milsims) were analyzed due to the inherently political and violent nature of the gameplay. To gather further research data we developed code to scrape areas of the milsim communities. The deliverables for the project were a community forum and YouTube scraper, cleaned data, visualizations, and sentiment analysis. We collected large datasets from both the community forums and YouTube, successfully cleaned the data, and did an analysis to create interesting visualizations. This did not just happen easily, we had to figure out how to: access data with page pagination, protect from unintentional SQL injections from online comments, as well as how to work around a site being migrated back in 2021. Additionally sentiment analysis was originally going to be conducted by the client but was delayed past the submission of our report and the handoff of our data, so we created our own analysis methods to produce interesting visualizations. Two of these visualizations showed us that the potentially extremist language on both platforms is very similar, both in word choice and frequency. This was interesting to us as it suggests the community guides the use of language much more than does the platform the community is on. During development we hit several problems mostly dealing with the forums site.
3.0 Introduction

3.1 Problem

Online gaming has increased in popularity in recent years, going from 2.03B gamers in 2015 to 3.09B in 2022 [2][30]. With it, new social communities have cropped up along with the games. These communities, particularly for hyper-realistic military simulation games (referred to as “milsims”), can employ hateful and extremist language relating to Nazism and other hateful ideologies. The issue is that we do not know exactly where and why this language appears, and there isn’t a lot of related research. Additionally, the information we’re looking at is scattered over hundreds of thousands of threads, on tens or possibly hundreds of different platforms. Even when the data is collected, it is messy. Plenty of excess whitespace, emojis, non-English characters, and random images, make analyzing it an arduous task.

3.2 Client and Motivation

Our client is Dr. James D. Ivory of the Virginia Tech Gaming and Media Effects Lab [4]. He is researching extremism, video games, simulations, and virtual environments. His motivation for this project is to use data preparation / curation, analysis, and reports to advance his understanding of extremist behavior in online game communities – which is associated with real-life extremist behavior, including high-profile criminal activity. This is the third CS4624 project for which he has been a client; our predecessors completed a similar assignment on Arma 3 [1]. So an additional motivation is to expand the number of games that have been researched to allow for more evidence to support claims made for gaming communities as a whole, not just Arma 3 [5] or War Thunder [6].

3.3 General Approach

Our approach was separated into 3 parts: scraping, cleaning, and analyzing. For the scraping we had two targets: YouTube [7] video comments and War Thunder Forums [8] (referred to as Forums at times). The previous work left to us by last year’s team included a JavaScript file that can scrape YouTube video IDs when provided with a video tag. It utilizes the free YouTube developer API [9] which gives powerful tools to scrape data from the site and has very reasonable daily request limits. We adapted parts of this script to better fit our purpose, and
switched the target from Arma 3 to War Thunder play-through videos. Additionally we made a separate script that would take in the video IDs and scrape the comments. The War Thunder Forum unfortunately did not provide a public API. They did however allow requests to their backend from any IP address. This allows us to skip using a headful browser bot like Selenium [10], which would require scraping data a lot closer to human speed as it steps through all parts of the site like a human would. Instead we can utilize Python’s request library to pull down the forum threads HTML for analysis. Using another Python library, lxml [11], we can parse through the retrieved HTML much like you can with JavaScript in the DOM. This makes it fairly simple to write functions to extract specific data from the HTML. One issue with this approach is the site paginates its data, only showing more comments when the user scrolls to the bottom of the page. To get around this we figured out the URL can have a number appended to the end that will dictate what comment index we want, and will return the 30 comments after that. This allowed us to retrieve a thread of 300 comments in 10 requests, 600 in 20, and so on. All this data is assembled into an object and then stored in a SQLite3 database [12]. We chose SQLite3 due to its light overhead, terrific filtering, and query searching. The lack of complex relation support did not hold us back, as we only had very simple table relations. For cleaning we utilized modified versions of our client’s cleaning methods, which were made via Python scripts, and stored the cleaned data in a new database table. The analysis stage would use sentiment analysis with an extremist language dictionary [29] provided by the Anti-Defamation League (ADL), which can classify a comment as extremist or non-extremist, and was to be conducted by the client. This was the initial form of analysis we were going to generate visualizations off of. Unfortunately the client had issues with their sentiment analysis software so we ended up doing our own analysis; that was not a full sentiment analysis but instead involved keyword searches and frequency analyses. The client will hopefully conduct the full sentiment analysis with our data sometime after the sunset of this project. After we analyzed the sorted data, we used Python, Pandas [14], and Plotly [15] to create visualizations relating to the information.
4.0 Requirements

We were tasked with providing 60,000 data points from the War Thunder Forums; 60,000 data points from War Thunder YouTube videos; a War Thunder Forum scraper; a YouTube comment scraper; sentiment analysis with the client; and suitable visualizations.

4.1 War Thunder Forums Scraper

We developed an easy to use script that will gather all comments and metadata (user, date/time, post ID) from a randomized list of chat threads, automatically storing the data to our database. We also wanted to be able to update our data as threads are constantly being added to, especially if they’re current. So if the script is executing a thread that’s already been collected from, it should not unnecessarily duplicate the data, rather only save the new comments on the thread.

Along with this we include developer documentation to explain how it works, areas where it could be improved, future directions, and more to help future developers and researchers get the most out of the script.

4.2 YouTube Scraper

We created a Google Sheet Apps Script file that when run reads in video IDs and copies the comments and sub-comments over to our Google Sheet to later be exported to CSV files. This script works along with a script provided by last year’s team that scrapes video IDs based on a video tag. Using the scripts together we were able to effectively find videos relating to War Thunder and scrape their comments with relative ease. Additionally we created a script to export the many sub-sheets in the Google Sheet file to CSV files. Another script reads them into our database to be processed just like the Forums data.

See Section 9.6 for documentation and run instructions.
4.3 Data Collected

The data we collected both from YouTube and War Thunder Forums was processed to be given to the client in a cleaned and analysis ready state. It is provided in structured CSV files as well as a database. Table 1 gives numbers for our deliverable data.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Goal Comments</th>
<th>Actual Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forums</td>
<td>60,000</td>
<td>73,595</td>
</tr>
<tr>
<td>YouTube</td>
<td>60,000</td>
<td>122,301</td>
</tr>
</tbody>
</table>

Table 1: Scraped Comments Goal vs. Actual

As is shown in Table 1 you can see we produced more comments than what was originally expected for both platforms. Part of this was due to the metrics being based off last year’s project and the differences in our implementation, particularly our change to fully scrape Forum threads to completion rather than only taking the top 100 comments.

4.4 Sentiment Analysis Results

Due to setbacks with the client’s sentiment analysis scripts we were unable to work with them to do analysis this way. Instead we performed our own analysis to create visualizations. This analysis was much less exact, more just checking if any words from our extremist dictionary were present in the comments and what those words were. Additionally it’s important to note that through this method we are not necessarily calling comments with these words extreme. The words themselves are just good indicators of a possibly extremist post. For example “patriot” is a word in our dictionary that can be widely used and is not extreme; it just also happens to appear in a lot of extremist language as well. The client has the data and will be able to conduct the actual sentiment analysis later on and draw more conclusive results from that.

4.5 Visualizations of Data

Our visualizations are made from comparing dates with comment rates, ratios between total comments and the number of potentially extreme comments, as well as word clouds of most
commonly used potentially extremist keywords. They are provided in the form of word clouds and bar charts.
5.0 Design

For our design we wanted to fully separate the scraper, cleaning, analysis, and visualizations into completely different sections. So we segmented our scraper into multiple separate scripts. This was to fit the use cases of a web scraper. Some scripts will be in different programming languages as well as for different domains, so it makes sense to divide our scrapers into a YouTube scraper and a War Thunder Forum scraper. For the War Thunder Forums some portions that get scraped will not change often and don’t need to be re-scraped during every execution, so we divided it up. There are separate scripts for finding all categories and threads, randomizing the threads to scrape, and the actual scraping of posts. For the YouTube scraper we divided it up into finding video IDs with specific tags, scraping comments from those videos, and exporting from Google Sheets to CSV. This division also had the added benefits of being easy to test a small portion at a time, as well as having unexpected errors only resetting one portion of the process – not the entire thing. The cleaning was virtually the same for both platforms as the data was quite similarly structured. Analysis and visualizations were also virtually the same between the two domains.
6.0 Implementation

The implementation of our project consisted of visualizations after scraping and cleaning. For the scraping we used both Python and Google Script. Google Script was used mainly for scraping the YouTube side of our project, while Python was used for the War Thunder Forum scraping, the cleaning of our data, and the handling of our data from our database. The visualizations we created used our data and Python.

6.1 Scraping

The War Thunder Forum scraper was made via Python. The direction we went with our scraping relied on if we needed to use Selenium to scrape the info or if we could do it based on regular HTTP requests. Thankfully it turned out to be doable via requests, allowing us to have a simpler and faster design. We have a script that gathers all the categories and their subsequent threads on the War Thunder Forums. This will not need to be run as much, as new categories and threads are created much less frequently than comments themselves. It will ping a specific forum URL with increasing indexes, and saves all URLs and category names that don't return a 404 page. Then it uses all the saved URLs as well as increasing page indexes in the URL and request headers to find all connected threads. Then we have the thread randomizer that will take a number of target threads as input and randomly mark that many as viable for scraping, printing out the total number of comments all those threads are expected to produce. This script was necessary to insure data validity and prove we did not cherry pick particular threads to find more extreme data. The final step is the actual post or comment scraper. It takes a list of those marked threads and begins running HTTP requests pulling in the data and restructuring it with lxml to fit our database.

The YouTube scraper is made using Google Apps Script (Google Script) [26] with Google Sheets (Sheets) [27]. It has a script that will take a tag and scrape the top 100 videos containing that tag. It populates a Sheet with the video ID, title, and other relevant metadata. The second script will run through the video ID column, using the value to build a URL to the specified video. It uses that URL to scrape the top 100 comments, making sure to capture every comment's sub-comments. It stores the username, comment, and other relevant data in a new sub-sheet or page in the Google Sheet. It was modified from Randie Pathirage's YouTube comment script [25]. The title of the sub-sheet is the video ID that was scraped. The final Google Script is only
slightly modified from the original created by Soham [16] and it converts all sub-sheets in a Sheet to individual CSV files, exporting them as a zipped Google Drive file. This was necessary as we realized Google Sheets only lets you export one sub-sheet to CSV at a time and we had over 600 sub-sheets that would need to be exported one by one. Once we had the downloaded zip file we used a Python script to read in the CSV files and store both the videos and comments in our database.

6.2 Cleaning

In order to clean the raw data that we scraped, we created a Python cleaning script. Unlike the previous team, we decided to clean straight from our database rather than from CSV files as it was quicker and allowed us to easily reuse a lot of code. We started with adding user input to select which type of data the user wanted to clean since our YouTube and Forum data are stored differently in our database. Then using the user's input it starts gathering the forum data or the YouTube data. Then we started cleaning the comments by first removing any leading or ending white spaces such as tab, newline, and other carriage return and line feed characters. After that we removed any non-ASCII characters within each comment. After the cleaning was completed, we exported the clean comments to CSV files, one CSV file for each dataset. This was due to how our client wanted to receive the cleaned data.

6.3 Visualizations

For our visualizations and analysis, we create Python visualizations scripts for different types of visualizations we decided to use. We wanted to create visualizations that we felt would best represent the data that we have scraped for our client as well as the purpose behind the scraping of the data. Hence, we decided to create visualizations that used the comments we scraped and compared them to the previous team's dictionary text file, which is a text file full of possible extremist language. In order to do this, we started with creating three Python scripts, one to compare YouTubes data with the dictionary file over a timeline, another to do the same for the Forum, and the last one to use both YouTube and Forum data and create two graphs, one for YouTube and one for the War Thunder Forum, of the total comments scraped versus the number of comments that match or contained words or phrases from the dictionary file. For the visualizations that used time, we collected the data from our database, as our client did not want the times in the cleaned CSV files. For the visualizations that did not need the time, we collected the comments from the cleaned CSV files. After collecting the data needed for the
visualizations, we then compared each word or phrase in the dictionary file to each comment gathered and made a count if anything in the dictionary matched in a comment. We had to be careful to not do partial word matches, for example “ab” is in our dictionary and that could match a word like “cab” which we don’t want. Then we created the visualizations using matplotlib to create our graphs.

Figure 1: Total YouTube Comments vs. YouTube Comments Matching Dictionary Words
Figures 1 and 2 show the number of YouTube and Forum comments, and show the number of those comments that match words or phrases in the dictionary file supplied to us by the previous team. We can tell from the graphs that even with the lower number of total comments, that the forum has more comments that match the dictionary compared to the YouTube comments. We can not say anything conclusive about these findings as we found that YouTube comments are more common to have bots posting comments that are simply comments that promote their accounts. Further, our analysis simply detected the presence of these words, not what the author used them for, which a more proper sentiment analysis could determine.
Figure 3: Forum Comments Matching Dictionary Over Time
Figures 3 and 4 show the number of comments that match any phrase or words in the dictionary file over time. We can see that the YouTube data we gathered has a much longer timeline than the forum comments. This is due to the War Thunder Forum being relocated to another website at the beginning of 2022. We only gathered comments from the new War Thunder Forum.

After the original scripts, we decided to create two more visualizations that showed our data in a different way. Using the wordcloud [17] Python package made this fairly easy. We just had to construct a list of words and it would show the word cloud based on each unique word’s frequency in the list. To do this we ran both the YouTube and Forum data through a program that would detect each instance of a word in our extremist dictionary and add it to a list. Once this was done we built the wordclouds using that list and obtained results suitable for visualizations.

Figure 5: War Thunder Forum Word Cloud
Figures 5 and 6 show a Word Cloud for each of our datasets, YouTube and War Thunder Forum. For these visualizations we wanted to see and compare which extremist keywords were commonly found in each domain. We were able to find and show how a lot of the potentially extremist words are similar between the two platforms, which surprised members of our team. Some of us figured users would communicate differently on the platforms. However, as shown by the word clouds, it seems that the same language goes across the two domains, even keeping relatively similar frequencies. For example if you look at the word “aim” in both Figures 5 and 6 you can see they are nearly the same size. The same is true for “delta”, “patriot”, “fear” and a few others. Keep in mind the word cloud is generated only from potentially extremist words, so the ratio of extremist language to normal language could be much different on the platforms.
Figure 7: Number of Forum Posts on the Forum on a Leak Day

Figure 7 [28] shows the dates on which confirmed leaks occurred on the War Thunder Forum, and we were seeing how many posts we captured that were posted on the same day as the leaks. The leaked documents are as follows starting with the first date in the graph, August 31, 2023: Eurofighter Typhoon DA7 fighter jet, Lockheed F-117 Nighthawk stealth attack aircraft, Boeing AH-64D Apache helicopter, Norinco VT-4 tank, and M2A2 Bradley IFV. With about 150 posts a day on average that we captured, we can see that three of the five leaks went well over the daily average. So, perhaps more interest was sparked on the site. Also important to note is that the actual messages relating to the leak were most likely scrubbed entirely from the site, so actual numbers could possibly be much higher.
7.0 Testing/Evaluation/Assessment

7.1 Testing

For our first iteration of testing we simply printed out to the terminal what we were able to gather from the scraper. Figure 8 shows an example posted object.

![Example of War Thunder Post Python Object](image)

During the first iteration we were not able to store the data as we were just trying to see what data was found, and how much we were able to request at once. We then used that information to develop our next iteration.

Iteration two allowed us to test what we gathered by printing it again to the terminal and cross checking that the data went into the database that we created. We manually checked to make sure that duplicates did not go into the database, and checked to see what type of data was causing errors.

Iteration 3 changed how we receive our testing data, as we changed the input and output format. Instead of changing the script’s variable for thread ID, we accepted user input as a thread ID; see Figure 9. Additionally, instead of printing out to the terminal, we printed the errors to a text file. We also printed what current posts were being scraped as we have to iterate through a forum discussion and cannot do the entire discussion at once. This showed the progress to the user. We had error messages regarding failing database inserts as some characters in the data were causing us issues, mainly double quotes (" ").
Iteration 4 was our final iteration of the main post scraper and testing. It now pulls all threads marked for scraping from the database instead of taking user input. We also switched back to printing errors to the terminal as the number of errors decreased dramatically. The errors decreased because we implemented SQL injection prevention.

The testing of the YouTube scraper involved running the scraper for a few iterations so the quota was not empty. To check the scraped information was correct, we manually searched the video based on the video ID to see if the scrapped information was correct. With the comments, the same idea was involved: search from the video ID, and check to see if the comment appears.

7.2 Evaluation

Table 2 gives some performance metrics for our scraping, cleaning, randomizing, exporting, and storing scripts. Unfortunately, we did not have the foresight to time our scraping of data we turned over to the client. We did that prior to adding an evaluation section. With the variation of public online data it would be very difficult to recreate the exact same scraping of data to get timed statistics on it. While this is unfortunate we decided we could still time the processes; it would just need to be on a similar yet new data set. So Table 2 does not show the time it took to scrape the data for our client, it shows the time it took to scrape a similar amount of data while doing all the processes exactly the same. The threads, videos, and comments it scraped are mostly different, but the way we scraped it was the same. This allows us to have a general estimate of how long these scripts took to generate our clients data.

The device running these scripts is a MacBook Air with a M1 chip, 8 GB RAM, and 8 cores.
<table>
<thead>
<tr>
<th>Script + Input</th>
<th>Execution Time</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>category-thread-id-scraper.py</td>
<td>19 min 44 seconds</td>
<td>The following describes what was added to the DB: 69 categories; 20,327 threads; 500,492 potential posts (this is just a sum of the post count each thread has, not the actual posts being scraped)</td>
</tr>
<tr>
<td>randomizeThreads.py set to 2,750</td>
<td>3.269 seconds</td>
<td>2,750 threads marked for scraping; 64,699 potential posts from the marked threads</td>
</tr>
<tr>
<td>threads</td>
<td></td>
<td></td>
</tr>
<tr>
<td>post-scraper.py</td>
<td>1 hour 39 minutes and 28</td>
<td>64,740 posts scraped (you can see 41 posts were added since we got the thread’s post count.)</td>
</tr>
<tr>
<td>seconds</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cleaning.py for Forum posts</td>
<td>2.093 seconds</td>
<td>64,740 posts cleaned</td>
</tr>
<tr>
<td>youtube-video-scraper.gs</td>
<td>1 min 50 seconds</td>
<td>568 videos scraped</td>
</tr>
<tr>
<td>youtube-comment-scraper.gs</td>
<td>49 min 18 seconds</td>
<td>568 videos giving 106,497 comments</td>
</tr>
<tr>
<td>export.gs</td>
<td>14 min 6 seconds</td>
<td>Comments for 568 different videos exported to CSV</td>
</tr>
<tr>
<td>export.csv</td>
<td></td>
<td></td>
</tr>
<tr>
<td>youtube-db-importer.py</td>
<td>58 seconds</td>
<td>Transferred 568 videos and 106,497 comments from CSV to database</td>
</tr>
<tr>
<td>cleaning.py for YouTube comments</td>
<td>1.333 seconds</td>
<td>106,497 comments cleaned</td>
</tr>
</tbody>
</table>

Table 2: Execution times of scripts
*Please note the time for youtube-comment-scraper.gs is a combination of several run times of the script. Due to YouTube’s token limit with their API you can’t make this many requests at once. The process either is done over 2-4 days or 2-3 different accounts and 1-2 days.

Table 2 shows the running time of every part of our project. The longest running time is 1 hour and 39 minutes for post-scraper.py. We believe this higher runtime is mostly due to the pages only returning 30 comments at a time. This can turn a 90 comment thread into 3 separate requests, and each request is the longest part of the process. Additionally it gathers less data than the youtube-comment-scraper.gs but in more time. This seems fairly reasonable as we custom built all the interactions with the War Thunder Forums. On the other hand, with YouTube we get to use the YouTube API that has been worked on by multiple engineers for years to provide efficient and powerful access to their data. It is to be expected that a third party scraper would perform less efficiently than an official API.

For the actual data we sent the client, Table 3 gives some statistics on what we ended up with.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Categories</td>
<td>69</td>
</tr>
<tr>
<td>Threads</td>
<td>18,150</td>
</tr>
<tr>
<td>Total Forum Thread Views</td>
<td>14,225,662</td>
</tr>
<tr>
<td>Average Thread Per Category</td>
<td>263</td>
</tr>
<tr>
<td>Total Potential Posts</td>
<td>448,486</td>
</tr>
<tr>
<td>Average Post Per Thread</td>
<td>24.7</td>
</tr>
<tr>
<td>Scrapped Threads</td>
<td>2,750</td>
</tr>
<tr>
<td>Posts</td>
<td>73,596</td>
</tr>
<tr>
<td>% of Forum Posts Captured</td>
<td>16.4%</td>
</tr>
<tr>
<td>Videos</td>
<td>570</td>
</tr>
<tr>
<td>Scrapped Video Comments</td>
<td>122,301</td>
</tr>
<tr>
<td>Total YouTube Video Views</td>
<td>335,569,498</td>
</tr>
<tr>
<td>Total YouTube Video Likes</td>
<td>10,274,589</td>
</tr>
<tr>
<td>Metric</td>
<td>Value</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>Total YouTube Comments</td>
<td>348,741</td>
</tr>
<tr>
<td>% of YouTube Comments Captured</td>
<td>35.07%</td>
</tr>
<tr>
<td>Total Database Size</td>
<td>64 MB</td>
</tr>
<tr>
<td>Total Forum CSV Size</td>
<td>56.1 MB</td>
</tr>
<tr>
<td>Total YouTube CSV Size</td>
<td>12.8 MB</td>
</tr>
</tbody>
</table>

Table 3: Metric Data

One cool aspect of looking at Table 3 is the realization there’s so much data that we now know how to get, but just didn’t retrieve. With the scraper system in place it really is now just a matter of how long you allow your computer to run for. We scraped 2,750 threads, but we know there are about 18,150 threads total. Thus, we have only captured 16.4% of the known post count, which has only since gone up. If a user really wanted every public post on the Forums they could just mark all threads as scrapable manually rather than using randomizeThreads.py, and let their computer run, making sure they have enough storage as well.

It’s a bit less simple with YouTube as we’re bound by their API limits, so scraping more data is actually still more work for the user as opposed to the Forums. But still it wouldn’t be a ton of extra work, more just making sure you click “run” on every day that your tokens reset.

Overall the total size of the files is much less than we thought initially. It’s interesting that the entire database with more info than what’s in the CSVs is smaller than the two CSVs added together. That really shows off the efficiency of a database. Furthermore you can see the Forum’s data is multiple times the size of YouTube’s. This shows that at least for the War Thunder community, long posts/comments are more for the Forums than for YouTube.

7.3 Assessment

For assessing our scraper’s performance, there are many variables at play, mostly including regarding the internet connection and host machine. We just want a rough ballpark to tell if we’re way slower or faster than a conventional scraper. In a medium article [23] written by user vladkens they use their own Twitter scraper [24] to scrape 5,000 tweets from Twitter. They
weren’t too specific about runtime but said it would take 5 to 10 minutes to scrape the 5,000 tweets.

Let’s compare our War Thunder Forum Scraper first:
We had 64,740 posts scraped in 1 hour 39 minutes 28 seconds. That’s 5,968 seconds. Our 64,740 posts is 12.948 times larger than the 5,000 tweets scraped by vladkens. On the lower end of 5 minutes if we scale his runtime to match our dataset size you get 3,884.4 seconds. On the higher end with a runtime of 10 minutes for 5,000 tweets it’s 7,768.8 seconds if their dataset was the same size as ours. You can see our runtime of 5,968 seconds is in-between the lower end of 3,884.4 seconds and the higher end of 7,768.8 seconds. This seems like a reasonable comparison for our custom Forum scraper, showing it operates about as fast as a Twitter scraper does for a similar dataset size.

Now for the YouTube Scraper:
We could not find good comparison data for YouTube scrapers because it’s more about the tokens than runtime. We were using a composite runtime and are unable to find similar runtime data on YouTube API projects. Nor does it feel fair to compare our API scraper to the Twitter vladkens 3rd party scraper (since due to changes in Twitter’s public API he couldn’t use it). We are however happy with the actual runtime; the limiting factor was the token constraint, which besides running multiple accounts at once we couldn’t bypass.
8.0 Users' Manual

8.1 Forum Manual

8.1.1 Setup

The application we built for scraping the Forum data is run via Python in the command line. Wherever the code has been downloaded, you should navigate to there, using the command line. Here’s how:

- **Mac**:
  - Open Spotlight (cmd + space) and search for “terminal”
  - Use “cd <path-to-directory>” for whatever directory you saved the file in

- **Windows**:
  - Press the Windows key and search “Command Prompt” or “CMD”
  - Use “cd <path-to-directory>” for whatever directory you saved the file in

You must also have Python 3 installed on your system (Python 2 might work but does not match our installation and has not been tested). Here’s a resource [18] on how to install Python 3.

Note: If you have both Python 3 and 2 installed you can use the command “python3” instead of “python” to make sure you’re using the correct version.

Additionally you’ll also need SQLite3 installed locally. Here’s a resource to help with installing it [19].

The database will already have our current data in it (skip to 8.1.2 if you don’t want to reset the database), so if you want to start from scratch, delete database.sql and create a new file with the same name in the same location. To create the database tables run this in the terminal in the GamesBehavior24 directory:

```bash
> python
> from Database import Database
> Database().create_database()
> quit()
```
8.1.2 Scrape Categories and Threads

Now with a fresh database you can begin by populating it with threads. Please run the following in the terminal at the GamesBehavior24 directory: `> python category-thread-id-scraper.py`

This will take about 10 to 20 minutes to fully complete. But once it’s done you should be able to see your categories and threads by running:

```
> sqlite3
> .open database.sql
> SELECT COUNT(*) FROM Categories;
> SELECT COUNT(*) FROM Potential_Thread;
```

When we ran this script we had 69 Categories and 18,150 Potential_Thread(s).

8.1.3 Randomize Target Threads

Now we need to randomize our target threads to not cherry pick data from particular threads. Skip this step if you wish to scrape all the site’s threads. Before running the randomizeThreads.py script, open it up in a text editor and change the value of `numOfThreadsToScrape` to equal whatever number you want. It should be the top most variable in the script directly under the imports. Currently it’s set to what we used which was 2750 threads. After that it’s ready to run: `> python randomizeThreads.py`

It will print out how many threads are marked for scraping and the total number of posts those threads say they have. Note that this number is just an estimate, depending on how long ago you scraped thread data, the categories and threads posts could have been removed or added changing the value of that number. In most cases we found we would get slightly more posts than expected due to new posts. To verify it marked the threads you can run:

```
> sqlite3
> .open database.sql
> SELECT * FROM Potential_Thread WHERE will_scrape = 1;
```

Feel free to rerun the randomizer as much as you want till you’re happy with the estimated number of threads. We had several runs that were estimated under our goal of 60,000 posts so we reran till one gave us over 70,000 posts. This variability is due to randomizing based on the
number of threads not the number of posts. Most threads have under 100 posts so you can get lower numbers of posts if those are randomly chosen en masse. Modifying it to be based on posts would be a more complex process, and there was very little need when you can just run the script a few times and get a good estimated value while still keeping things random.

8.1.4 Scrape Posts

Now that we have a randomized database we can easily run post-scraper.py to actually scrape the posts. Keep in mind this process will likely take at least an hour depending on how many threads you target. For over 60,000 posts it took us 1 hour and 44 minutes or so to complete.

Here's how you run it: ```python post-scraper.py```

```
Running scrape on posts [0 to 36]
Running scrape on posts [27 to 57]
Running scrape on posts [47 to 77]
Running scrape on posts [67 to 97]
Running scrape on posts [88 to 118]
Running scrape on posts [108 to 138]
Running scrape on posts [128 to 158]
Running scrape on posts [148 to 178]
Running scrape on posts [168 to 198]
Running scrape on posts [188 to 218]
Running scrape on posts [208 to 238]
Running scrape on posts [228 to 258]
Running scrape on posts [248 to 278]
Running scrape on posts [268 to 298]
Running scrape on posts [278 to 308]
Running scrape on posts [298 to 328]
Running scrape on posts [318 to 348]
Running scrape on posts [338 to 368]
Running scrape on posts [358 to 388]
Running scrape on posts [378 to 408]
Running scrape on posts [398 to 428]
Running scrape on posts [418 to 448]
Running scrape on posts [438 to 468]
```

Figure 10: Screenshot of the Scraper Script Executing

Figure 10 illustrates scraping a forum thread after running post-scraper.py. This is what the output should look like. It will then also tell you in the terminal when it switches over to a new thread. Any failed inserts will pop up in the console as well, but the majority of the failed inserts are when we’ve already inserted that comment, so the data is already saved and the failure doesn’t lose any information.
Currently, to browse the queries you need to use the SQLite3 command line or a visual database browser like DB Browser [20].

Below are some helpful queries to run to visualize your data:

The query below will give you all posts from one thread.

```
SELECT * FROM Post WHERE thread_id = 410;
```

![Figure 11: Query output 1](image1.png)

Your output should look similar to Figure 11.

The query below will give you all posts where the message includes ‘hi’.

```
SELECT * FROM Post WHERE message like '%hi%';
```

![Figure 12: Query output 2](image2.png)

Your output should look similar to Figure 12.

The query below will give you all distinct authors in a thread and how many times each was found in a post.

```
SELECT author, COUNT(*) FROM post WHERE thread_id = 3725 GROUP BY author ORDER BY COUNT(*) DESC;
```
Figure 13 is an example of the output you can expect to see. We are fairly sure Morvran is a developer for War Thunder as we constantly see him in nearly every thread with some of the highest post counts.

8.2 YouTube Manual

The YouTube script is designed to scrape videos and return information based on the video. The way in which to begin the scraping follows.

Figure 14: To enter the Google application script
Figure 14 shows how to enter the Google apps script. This allowed us to write scripts with built-in Google and YouTube APIs. From there, on the left side, there are libraries and services to add. Click on the + in the services section and search for the YouTube Data API, and add it. This will give access to all the API services for YouTube.

The scraper was split into 2 files. One to scrape the video information; see Figure 15. The second was to scrape the comments; see Figure 16.

```javascript
function scrapeVideoRow(items, search_query, activeSheet, offset) {
  var next = null;
  varoment = new Date(2017, 0, 1, 0, 0, 0);
  var count = 0;

  for (var i = offset; i < items.length; i++) {
    var videoId = items[i].id.videoId;
    var videoTitle = items[i].snippet.localized.title;
    var videoDescription = items[i].snippet.localized.description;
    var videoPublishDate = items[i].snippet.publishedAt;
    var videoThumb = items[i].snippet.thumbnails.high.url;

    activeSheet.appendRow([videoTitle, videoDescription, videoPublishDate, videoThumb]);
  }
}

function scrapeCommentRow(items, search_query, activeSheet, offset) {
  var next = null;
  varoment = new Date(2017, 0, 1, 0, 0, 0);
  var count = 0;

  for (var i = offset; i < items.length; i++) {
    var videoId = items[i].id.videoId;
    var videoTitle = items[i].snippet.localized.title;
    var videoDescription = items[i].snippet.localized.description;
    var videoPublishDate = items[i].snippet.publishedAt;
    var videoThumb = items[i].snippet.thumbnails.high.url;

    activeSheet.appendRow([videoTitle, videoDescription, videoPublishDate, videoThumb]);
  }
}
```

Figure 15: Screenshot of the video scraper
These 2 files work together. To begin, make sure the script runs the startVideos function first, because it will return an error if not. To make sure it will call startVideos first, make sure the dropdown box next to Debug at the top is set to startVideos. Then, when run, the video scraper will scrape random videos from a given search tag and place them in the same spreadsheet, with the first column being the video IDs. Once this is done, the comment scraper’s prerequisites are complete. Now you can run comments.gs [25]; make sure that the starting function is startComments, otherwise an error will occur. The comments scraper will parse through all of the video IDs from the video spreadsheet and create a new sheet with some of the comments from the video.
Since there wasn’t a way to mass export sheets with Google, we had to find a way to export them quickly. Therefore, we found the exporter.gs script [16] pictured in Figure 17. This allowed us to export all of the sheets from the 1 sheet, to each respective CSV sheet. With the data now in your Drive download it locally in a zip file, to move it to our main database to send for cleaning.
This file pictured in Figure 18 allowed us to consolidate all of the Sheets data in the database just like the War Thunder Forums scraping. To start the database import you need to unzip and drag the folder that was exported to Google Drive by exporter.gs [16] into the GamesBehavior24/YoutubeCSV/ directory. This folder when exported will be called “Capstone_Sheets.zip” by default. Once it’s in the correct directory you can run youtube-db-importer.py which will parse through all of the sheets and add it to the database with the correct format. Additionally it will first parse the video ID sheet in the folder, called “Videos.csv” by default, to populate the Videos table with the video information.
9.0 Developer's Manual

Figure 19: Overview Image of Forums Codebase

Figure 20: Overview Image of YouTube Codebase
Figure 19 shows the interaction between the files, database, and the War Thunder Forums site. Figure 20 shows the interaction between the files, database, Google Sheet, and YouTube’s API. The numbered arrows describe the order of interactions between the scripts. For example youtube-comment-scraper.gs pulls from Google Sheets, then YouTube, then adds data to Google Sheets in that order.

9.1 Inventory of All Files

Table 4 summarizes all of the files that we have used throughout this project by their name, what the file does, and what type of file it is.

<table>
<thead>
<tr>
<th>Filename</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>category-id-thread-scraper.py</td>
<td>Uses HTTP requests with URL parameters and header manipulations to get every public category and subsequent thread from the War Thunder Forums.</td>
<td>Python programming script</td>
</tr>
<tr>
<td>CleanedData/</td>
<td>The directory that the cleaned CSV files for both platforms is exported to.</td>
<td>Directory containing CSV files</td>
</tr>
<tr>
<td>cleaning.py</td>
<td>Cleans either War Thunder Forums or YouTube data by pulling from the database, removing whitespace, non-UNICODE characters, and exporting to CSV files.</td>
<td>Python programming script</td>
</tr>
<tr>
<td>Database.py</td>
<td>Provides functions to facilitate interaction between Python and the SQLite3 database, such as insertions and selections. Used by nearly every other Python file in the project.</td>
<td>Python programming script</td>
</tr>
<tr>
<td>database.sql</td>
<td>The database itself. Stores and orders the records of posts and threads. Accessed by Database.py or by the user directly to browse data.</td>
<td>SQL database file</td>
</tr>
<tr>
<td>export.gs</td>
<td>Exports all sub-sheets in the Google Sheet to</td>
<td>Google Script</td>
</tr>
<tr>
<td>File/Script Name</td>
<td>Description</td>
<td>File Type</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>individual CSVs</td>
<td>Combines individual CSVs and places them in the user’s Google Drive.</td>
<td>Programming script</td>
</tr>
<tr>
<td><code>post-scraper.py</code></td>
<td>Reads in War Thunder Forum threads that are marked to be scraped from the database, and will scrape each thread entirely of its posts and metadata, adding it to the database.</td>
<td>Python programming script</td>
</tr>
<tr>
<td><code>queries.txt</code></td>
<td>This is a file we've been adding to that just holds some handy SQL queries for looking at the data.</td>
<td>Plain text file</td>
</tr>
<tr>
<td><code>randomizeThreads.py</code></td>
<td>Takes input in the form of the number of threads you’d like to scrape. It then randomly selects that number of threads from the known threads, marks them to be scraped, and prints out the total estimated number of posts that scraping all of the threads would produce.</td>
<td>Python programming script</td>
</tr>
<tr>
<td><code>README.md</code></td>
<td>Guide on installing packages, setting up the database, and first running the code.</td>
<td>Markdown file</td>
</tr>
<tr>
<td><code>youtube-comment-scraper.gs</code></td>
<td>Uses the list of video IDs created by <code>youtube-video-scraper.gs</code> to pull the top 100 comments from each video ID. It also pulls all sub-comments from each of the top 100 comments. The commenter username, date, likes, message, etc. are stored in a new sub-sheet with the sheet name as the video ID.</td>
<td>Google Script programming script</td>
</tr>
<tr>
<td><code>youtube-db-importer.py</code></td>
<td>Responsible for pulling in unclean YouTube data CSV files and adding them to the database.</td>
<td>Python programming script</td>
</tr>
<tr>
<td><code>youtube-video-scraper.gs</code></td>
<td>Searches top 100 most popular YouTube videos for each tag in an array. So if you include two tags you'll get 200 videos. It scrapes the video ID, publication date, views, etc. and puts it into a</td>
<td>Google Script programming script</td>
</tr>
</tbody>
</table>
Google Sheet. Utilizes the YouTube API.

YouTubeCSV/ The directory that CSV files exported from Google Sheets are placed in to be read into the database. Directory containing directories of CSV files.

cleanedDataComparedToComments.py Creates visuals that show the total number of comments versus the total number of comments containing possibly extremist language in both War Thunder Forums and YouTube video data. Python programming script

forumDictionary.py Responsible for creating a War Thunder Forum visualization of their comments that have matching words or phrases in the dictionary text file over a timeline. Python programming script

youTubeDictionary.py Responsible for creating a YouTube visualization of their comments that have matching words or phrases in the dictionary text file over a timeline. Python programming script

Table 4: Table of all files

9.2 Overview: Data Shape and DB

Table 5 summarizes the database tables used for the War Thunder Forum data. “{pk}” means that it is a primary key and “{fk}” means that it is a foreign key.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>post_id</td>
<td>Int, {pk}</td>
</tr>
<tr>
<td>thread_id</td>
<td>Int, {fk}, {pk}</td>
</tr>
<tr>
<td>author</td>
<td>String</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>Int, {pk}</td>
</tr>
<tr>
<td>name</td>
<td>String</td>
</tr>
<tr>
<td>month</td>
<td>String</td>
</tr>
<tr>
<td>-------------</td>
<td>--------</td>
</tr>
<tr>
<td>day</td>
<td>Int</td>
</tr>
<tr>
<td>year</td>
<td>Int</td>
</tr>
<tr>
<td>time</td>
<td>String</td>
</tr>
<tr>
<td>message</td>
<td>String</td>
</tr>
</tbody>
</table>

**Potential_Thread**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>Int, {pk}</td>
</tr>
<tr>
<td>title</td>
<td>String</td>
</tr>
<tr>
<td>post_count</td>
<td>Int</td>
</tr>
<tr>
<td>reply_count</td>
<td>Int</td>
</tr>
<tr>
<td>created_at</td>
<td>String</td>
</tr>
<tr>
<td>last_posted_at</td>
<td>String</td>
</tr>
<tr>
<td>views</td>
<td>Int</td>
</tr>
<tr>
<td>category_id</td>
<td>Int, {fk}</td>
</tr>
<tr>
<td>will_scrape</td>
<td>Int</td>
</tr>
</tbody>
</table>

**Categories**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>Int, {pk}</td>
</tr>
<tr>
<td>title</td>
<td>String</td>
</tr>
</tbody>
</table>

Table 5: Forum Database Tables
Table 6 summarizes the database tables used for the YouTube data. “{pk}” means that it is a primary key and “{fk}” means that it is a foreign key.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>Int, {pk}</td>
<td>name</td>
<td>Int, {pk}</td>
</tr>
<tr>
<td>title</td>
<td>String</td>
<td>comment</td>
<td>String</td>
</tr>
<tr>
<td>timestamp</td>
<td>String</td>
<td>timestamp</td>
<td>String, {pk}</td>
</tr>
<tr>
<td>views</td>
<td>Int</td>
<td>likes</td>
<td>Int</td>
</tr>
<tr>
<td>likes</td>
<td>Int</td>
<td>replies</td>
<td>Int</td>
</tr>
<tr>
<td>comments</td>
<td>Int</td>
<td>video_id</td>
<td>String, {pk}, {fk}</td>
</tr>
</tbody>
</table>

Table 6: YouTube Database Tables

The data in Python keeps the same information as the database tables for ease of access and insertion. For example a Video tuple from the database will be represented in the program as a Python object structured like so:

```python
video = {"id":1, "title": "1", "timestamp": "05-08-2023:11:12:13", "views": 0, "likes": 0, "comments": 0}
```

9.4 Overview: Forum Scraping Process

This section describes how we do the Forum scraping. The same method is used for Categories and Threads with a few modifications, but they were based on post-scraper.py so we will just do an in-depth look at that file rather than all the very similar files. Table 7 gives a detailed view of how the post-scraper.py works and the functions involved.
<table>
<thead>
<tr>
<th>Description</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 21 gives the three lines that kick off our Python script. We take</td>
<td><img src="image" alt="Figure 21: Kickoff Python Code" /></td>
</tr>
<tr>
<td>input from the database threads marked to be scraped, scrape, and save to</td>
<td></td>
</tr>
<tr>
<td>the database.</td>
<td></td>
</tr>
<tr>
<td>Figure 22 is our main scraping thread and perhaps the least</td>
<td><img src="image" alt="Figure 22: scrape_thread Python Function" /></td>
</tr>
<tr>
<td>understandable portion of the codebase. What it does is take a</td>
<td></td>
</tr>
<tr>
<td>thread_id, and check whether it exists. If it does, it starts at the highest</td>
<td></td>
</tr>
<tr>
<td>index we’ve scraped so far; if not, it starts at index 0. It scrapes one</td>
<td></td>
</tr>
<tr>
<td>slice at the index until the post_ids stop growing, meaning we’ve hit the</td>
<td></td>
</tr>
<tr>
<td>end of the comments on a thread.</td>
<td></td>
</tr>
<tr>
<td>In Figure 23 the scrape one slice function will disassemble a single</td>
<td><img src="image" alt="Figure 23: scrape_one_slice_at_index Python Function" /></td>
</tr>
<tr>
<td>HTML response and return all the posts it constructs from it to the</td>
<td></td>
</tr>
<tr>
<td>scrape_thread function to be stored later.</td>
<td></td>
</tr>
</tbody>
</table>
Figure 24 is the HTML request function that will request posts at the index provided. This includes all the page’s HTML which will need to be filtered through. You can see on line 3 we put the thread ID and the index i to get past the page, not just returning all posts.

```
1 # gets the html from the forum thread id at index
2 def get_html_forum_thread_at_index(id, i=0):
3     req = requests.get
5             {id}{/} + str(i) if i != 0 else ")"
6     return req.text
```

Figure 24: get_html_forum_thread_at_index Python Function

This function Figure 25 makes use of the lxml library to convert raw HTML to a tree, allowing us to navigate it much like you can the DOM tree in JavaScript. Thus, we can now parse based on class, ID, or HTML tag type!

```
1 # generates a lxml tree for easier parsing
2 def make_lxml_tree(res):
3     return lxml.html.document_fromstring(res)
```

Figure 25: make_lxml_tree Python Function

When given our newly created tree, this function in Figure 26 will return an array of trees, all of which are the individual posts, further breaking down the HTML to parse through.

```
1 # returns all individual post trees
2 def get_post_trees(tree):
3     return tree.xpath(  
4         '//div[@class="topic-body crawler-post"]')
```

Figure 26: get_post_trees Python Function
Figure 27 is one of our functions that we give an individual post tree to. You can see how it easily parses out the post_id and converts it to an integer before returning it. There are 3 more functions like this to get timestamp, message, and author. They operate in much the same way.

Figure 27: parse_post_id_from_post Python Function

Figure 28 is the function that calls our Database.py script to store the saved posts. It doesn’t need to check if the post already exists because that is handled in Database.py instead.

Figure 28: store_posts Python Function

Table 7: Forum Scraping Process

9.5 Overview: Database.py

Table 8 summarizes the functions provided by the Database.py file. We will keep it brief, as it’s a lot more general to databases and isn’t as specific for our project.

<table>
<thead>
<tr>
<th>Description</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>This function is for a general purpose, show me everything, use case. Typically used when we were just building the DB to verify things were added. It returns all posts stored in the DB.</td>
<td>select_all_from_post(none)</td>
</tr>
</tbody>
</table>
This function is critical for efficient updating of already scraped threads. When we detect that we're scraping a previously scraped thread we call this to get the index at which to begin scraping again, resulting in less duplicated data.

This function allows us to verify that a post does not exist before inserting it into the DB. It makes it so we reduce error outputs for duplicate entries.

This function allows us to check whether we've scraped a thread before, and thus need to start at a different index.

This function inserts a new thread into the Thread relation, marking it as a previously scraped thread in case we update the data in the future.

This function is the most used and is how we push new posts into the database. It takes a post object and parses its attributes into a SQL insert statement.

Checks if the database.sql file has tables already created and returns true or false based on that.

Creates the default tables in the database so that a user could delete all our data and create a brand new database to start fresh. It uses db_exists() to make sure and not overwrite an existing database, requiring the user to manually delete it.

Returns a sum of post_count on each thread that is marked for scraping. Used by randomizeThreads.py to tell the user how many posts they can expect before they run post-scraper.py.

Returns a list of all threads that are to be scraped by
post-scraper.py.

Given a thread ID it will mark it to be scraped. Used by randomizeThreads.py.

Run at the start of randomizeThreads.py to make sure we’re starting with zero scrapable threads prior to randomly marking them.

Used to store a new video comment to the database.

Table 8: Database.py functions

We did not cover every function inside the Database.py file. Most are extremely redundant particularly between Forum tables and YouTube tables as we’re doing the same thing just with a slightly different structure. The functions above are just some of the most critical that needed explanations. The remaining functions are very clear on how they work and some aren’t even called anywhere; they were just useful functions while testing and developing.

9.6 Overview: YouTube Scripts

Table 9 summarizes the functions in youtube-video-scraper.gs, youtube-comment-scraper.gs, and youtube-db-importer.py.
tag, search_query is the tag, sheet_index is what sheet it should be putting the data into. This function uses the YouTube API to search for videos within a date range. It pulls the most popular videos with that tag, loops through them extracting data about views, comments, etc. It then sets those values within the sheet indicated by the variable.

<table>
<thead>
<tr>
<th>search_query, sheet_index</th>
</tr>
</thead>
</table>

Start the comment scraper. You can set the starting index of the comments, for example if you have headers the index should be 2 so it doesn’t try scraping the string “Video ID” as an actual video ID.

<table>
<thead>
<tr>
<th>startComments()</th>
</tr>
</thead>
</table>

X is the index of the video ID on the Sheet. Y is the index of the column it’s pulling the IDs from, so in our case it’s always 1. This function pulls the ID off the Sheet, starts looping through 100 comments, pulling all sub-comments as well. It then creates a new Sheet with the video ID as the label and puts the comments within the sheet with proper headings.

<table>
<thead>
<tr>
<th>scrapeComments(x,y)</th>
</tr>
</thead>
</table>

This kicks off our YouTube CSV to database importer code. It searches all files in the YoutubeCSV directory and for each CSV folder it calls scrape_videos and scrape_comment_sheets.

<table>
<thead>
<tr>
<th>scrape_all_videos_in_folder()</th>
</tr>
</thead>
</table>

Reads from the video ID and title sheet and adds a new video to the database with the proper values.

<table>
<thead>
<tr>
<th>scrape_videos(file_name)</th>
</tr>
</thead>
</table>

Scrapes the actual comment data from the CSV and inputs them in the database, linking them via video ID to the added videos.

| scrape_comment_sheets(file_name) |

Table 9: YouTube Scraping Process
## 10.0 Lessons Learned

### 10.1 Timeline

Table 10 summarizes the milestones, on the right side of the table, that were completed by a specific date, on the left side of the table.

<table>
<thead>
<tr>
<th>Date</th>
<th>Milestones</th>
</tr>
</thead>
</table>
| February 14, 2024  | ● Complete Presentation 1  
                   | ● Finish research for scraping a forum  
                   | ● Develop scraper iterations 1 and 2  
                   | ● Set up unclean dataset storage   |
| February 29, 2024  | ● Develop scraper iteration 3  
                   | ● Start scraping data from forum  
                   | ● Finish Report 1  
                   | ● Start testing scraper            |
| March 20, 2024     | ● Get previous team’s code and understand it  
                   | ● Determine YouTube Video list  
                   | ● Start scraping YouTube Videos  
                   | ● Complete Presentation 2          |
| March 31, 2024     | ● Get cleaning script/create one  
                   | ● Finish scraping  
                   | ● Start cleaning the data  
                   | ● Create storage for cleaned data  |
| April 13, 2024     | ● Finish cleaning data  
                   | ● Determine what we want to visualize |
| April 28, 2024     | ● Start Sentiment Analysis with client  
                   | ● Finish Visualizations            |
Table 10: Timeline/Milestones

| May 3, 2024 | • Start Final report  
|            | • Finish Final report  
|            | • Complete Final Presentation  
|            | • Submit Capstone project to VTechWorks |

10.2 Problems and Solutions

Table 11 summarizes problems that we have run into while completing this project and how we solved that problem, or if that problem is a future work, then how we could solve that problem.

<table>
<thead>
<tr>
<th>Problems</th>
<th>Solutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>War Thunder Forum loads additional comments when the user scrolls, not just showing them all at once, making HTTP requests hard to get all the comments from.</td>
<td>We figured out that when we scroll further down in the website, the URL is appended to with an index. So, for example, if the webpage has post 100 to 130 loaded, the URL ends with “/100”, indicating that the starting index is post 100. We took advantage of this to scrape every single post from a thread without needing to step through the browser with a library like Selenium, resulting in our parsing and script being much simpler than it could have been otherwise.</td>
</tr>
<tr>
<td>No information on the max number of posts in a thread returned from a request. We didn’t know when to stop scraping a thread.</td>
<td>To solve the no max number of posts problem we just had to add some code complexity to fix it. It is the messiest point of the code base, but works really well. Essentially every time we make a request for new posts, we save the highest recorded post_id. We keep</td>
</tr>
<tr>
<td>Requesting higher and higher indexes of posts and if the highest recorded post_id is the same after a new request we know there are no longer new posts and thus we hit the max.</td>
<td>We now store the threads we scrape as well as the posts. When given a thread, we first check that it isn’t stored. If it is then we take the max post_id for that thread and use it as our starting index. This means if we have 10,000 posts from thread 3 and re-scrape it a week later, we will begin at 10,001 not 0. This saves us hundreds of useless requests and minutes of time.</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>Updating scraping data on an already scraped thread would just re-request the entire thread, wasting lots of time.</td>
<td>We used to just pull the &lt;p&gt; tags out of the message HTML, figuring that contained massive amounts of content. It turns out we were wrong, so now we take all inner HTML tags of the block &lt;p&gt; or otherwise and keep them in a plain text string. This makes it so we can capture any tag in the post and moves the problem more to our cleaning script than the scraper.</td>
</tr>
<tr>
<td>We were only able to scrape comments from the main thread, not replies to other comments or users. The variability in HTML tags used within the posts themselves prevented us from getting reply comments. Some messages have &lt;br&gt; tags, &lt;img&gt; tags, &lt;quote&gt; blocks, &lt;ul&gt;, etc. This makes it hard to correctly scrape every form of HTML.</td>
<td>A solution that could have helped earlier would have been to do more contextual research on the website prior to starting construction of the scraper. We did plenty of analysis on the functionality of the forums but even a few more Google searches about its history would have revealed that the site had migrated a few years ago. Now the only solution is to create another scraper specific</td>
</tr>
<tr>
<td>War Thunder Forums site being migrated [21] in 2021 resulting in no data prior to 2022.</td>
<td></td>
</tr>
</tbody>
</table>
10.4 Future Work

Suggested future work for our project mostly revolves around SQL queries and Database.py. Currently, every time we add a post, we open a connection, check if the post exists, close the connection, re-open a connection, add the post, close the connection, and move on to the next post. In the future it might be helpful to do a check on all the posts we’re adding at once to see if they exist, remove the ones that do exist, and then in another connection add all the new posts in one insert statement. This would most likely reduce our time inserting into the database. The reality right now though is that it’s by no means slow as is, and has yet to affect our ability to quickly push in new posts to the database. It also lets errors be more specific to the exact post we’re inserting, and one failed insertion will only mess up one post, and not the many other posts we were adding. So, there are some advantages and disadvantages to the way it’s currently being implemented, but certainly it is an area to consider improving upon in the future.

Another area that could be improved is multi-threading. Currently we run everything in one thread. There would be a clear increase in efficiency if multiple threads were making requests and scraping separate thread IDs, and a final thread was responsible for storing in the database. Or an alternative would be to have multiple threads work on the same exact thread ID instead of different ones, since they could make concurrent requests at different indexes to scrape an entire thread in a much quicker time frame. These additions add a large amount of complexity in terms of threads, locks, and managing data handoffs. This, combined with the fact that we had no issues with the scraping time of the application, suggest this might not be an addition worth making.

Depending on how much use this system is going to have it would eventually make sense to use a cloud database rather than SQLite3 so that the database file doesn’t upload directly to GitHub and trigger its file size warnings.
Other future work that is not related to our current system would be to scrape more data on various different milsims through their forums and YouTube videos.

11.0 Acknowledgements

**Dr. James D. Ivory (jivory@vt.edu)**, our main client for the project, as he was very helpful in creating the original goal for our project and giving us a great learning experience. Additionally, he collaborated with us to refine the scope and identify the essential components of our capstone project.

**Michael Senters (michaels22@vt.edu)**, the graduate student researcher for Dr. Ivory with whom our team worked. His guidance in delineating project requirements, outlining the initial steps, and articulating the desired technical outcomes was instrumental in steering our efforts in the right direction.

**Dr. Edward Fox (fox@vt.edu)**, our professor for CS 4624, as he gave us the opportunity to select this project and made sure the communication between us and the client was established. Additionally he worked with us to improve team cohesiveness and prevented a multitude of errors with his thorough inspections and commenting on our submissions.
12.0 References


https://docs.google.com/spreadsheets/create


[29] Anti-Defamation League, “Hate on Display”, 2024, Accessed April 2024
https://www.adl.org/resources/hate-symbols/search?f%5B0%5D=topic%3A1703