Technical Report - Final Project DS 5110: Surveying 1 Million Diverse Wildchat Conversations: In the Wild Use Cases of ChatGPT

Team Members: Jason Leung and Sharan Giri
Khoury College of Computer Sciences
Data Science Program
leung.jaso@northeastern.edu, giri.sha@northeastern.edu

November 27, 2024

Contents

1 Introduction

This technical report presents the findings and methodologies of our DS5110 final project investigating the WildCHAT dataset, a wide-ranging corpus comprising 2.5+ million chat interactions between anonymized users and chatGPT conversation bots. The goals of the project are multi-faceted and include pre-processing and wrangling the dataset for NLP analysis, investigating journalism/plagiarism related conversations as well as coding related conversations, and designing and building a Spark and Flask enabled keyword-based search mechanism that enables the retrieval of similar chat responses based on specific filter categories. This Flask user interface also facilitates a more granular analysis of user-bot dialogues in addition to displaying real-time text summarization of the search result ChatGPT conversations utilizing a BART transformer model.

The overall motivation of the project was to take advantage of this extremely rich and fresh Wildchat dataset as the field and technology of Large Language Models (LLM) continues to explode in the recent day and age. We aimed to learn and implement a variety of natural language processing techniques using different aspects of this dataset and we wanted to end up with a user-friendly informative search interface that could be demo'ed to our DS5110 class such that they can do quick real-time investigations of keywords of their own and read through and understand their results all within the duration of our final presentation. Given the time limitations of the two month duration of our project, we necessarily needed to focus on this core implementation and functionality as well as exploring just 2 of the many conversation categories of interest, programming and coding related ChatGPT conversations as well as journalism/plagiarism conversations. We reproduce some of the results of a well designed case study of generative AI's use in journalism.

The WildCHAT dataset was collected and curated by researchers at Cornell's Allen Institute of Artificial Intelligence, the University of Southern California and the University of Washington and contains over 1 million ChatGPT model 3.5 and model 4 conversations from most major countries of the world captured over the course of 2023 through early 2024. We asked for and received authorization to use the restricted access full toxic dataset for our project, with the aim of investigating jailbreaking conversations, where the user is trying to break the self-regulating behavior of ChatGPT to output something it normally has rules against doing, for example in terms of hacking cryptography code or of a sexual nature or even how to construct a bomb. However, the complexity and time investment of the jailbreaking topic was too much given our project timeframe of two months. Instead, we chose to investigate two straightfoward conversation categories. The first was journalism/plagiarism. We were able to reproduce some of the results of the research paper "Breaking News: Case Studies of Generative AI's Use in Journalism" in finding all of the conversations of a Mediterranean based journalist and his or her misuse of ChatGPT to either plagiarize other news outlets' articles, rewrite entire articles to report on press releases or just to rewrite articles based on primary sources. Lastly, we investigated programming/coding related conversations, especially conversations where ChatGPT was given a prompt to code a full program in a specified programming language. We compare a varied range of programming centric conversations and the in the wild output that ChatGPT gives based on user coding prompts versus a state of the art programming specialized LLM model, the DeepSeek-Coder-V2, which overall outperforms ChatGPT 4.0 and especially 3.5 on standard benchmarks.

2 Literature Review

The scientific literature centered on ChatGPT and the different use cases of ChatGPT, for recreation, education, programming, journalism or in many other industries, is still emerging. The two seminal research papers that we base our final project on are "Wildchat: 1m chatGPT Interaction Logs in the Wild." which introduces the whole Wildchat dataset to the public, as well as a subsequent paper by the same research group titled "Wild- Vis: Open Source Visualizer for Million-Scale Chat Logs in the Wild.".

Earlier in 2024, this research group presented Wildchat as the largest and most diverse publicly available large language model conversation corpus in the world and describes its diversity as well as its comparison to other LLM datasets including Alpaca, ShareGPT and the next "best" dataset LMSYS-Chat-1M. One of the greatest differentiators between Wildchat and LMSYS-Chat is the diversity of languages in Wildchat with almost half of all conversations having a majority of text in a language other than English, as compared to LMSYS-Chat which is predominantly English. The conversations of Wildchat also average a quite long 2.54 turns of user prompt and chatbot response and contain a long tail of conversations with 10, 20 and even 50 or 100 turns. Unfortunately, the scope of our current project is such that we were not able to explore any non-English conversations. All ChatGPT users of the research groups user interface must actively consent to their data collection, use and sharing agreement before being able to use their version of ChatGPT.

The group's second paper, Wild-Vis, an open source visualizer of Wildchat, serves as the original inspiration of our keyword search Flask interface. They are able to search through the full Wildchat dataset and display in real-time a text embedding clustering of the most relevant conversations pertinent to the searched keywords as well as a visual representation of the spread of most relevant conversations adjacent to this search term. Users can navigate from conversation to conversation and pull up full details and text of each conversation. Their Wild-Vis visualization tool also is also able to display the full spread of a search term in the full text embedding space of Wildchat and compare it to the clustering spread of for example the LMSYS-Chat dataset. While, the back-end real-time NLP processing and visualization functionality of Wild-Vis was beyond the scope of our project, these research papers did highlight to us that the investigation and empirical study of large language models is very much an area of ongoing research and that it is still very much an area ripe for discovery.

Wildchat and ChatGPT forms one half of the background of our project. But the other half was researching, learning and implementing varied natural language processing techniques. Initially, we found an efficient topic modelling algorithm from the FASTopic python library. This was introduced in their paper: "FASTopic: A Fast, Adaptive, Stable, and Transferable Topic Modeling Paradigm." In the middle of our project, our goal was to create an upgraded version of the simple keyword search functionality but using topic modelling to generate the core topics that every single conversation belongs to. When a user would then input a keyword to search, it would not be limited to pulling up every conversation that solely contained that exact keyword in its text but also those conversations where the word via the topic modelling pre-processing is considered part of the main topic of that conversation. However, even though the FASTopic authors designed their topic modelling algorithm to be faster than the conventional mechanism, by representing topic clustering mathematically as a transport plan algorithm, the reality was that the running time and memory constraints of FASTopic were such that it could only be run on a maximum of 3,000 conversations at a time and even at this scale processing would take over 30 minutes at a time. These time constraints were 3-4 orders of magnitude slower than we had hoped for and meant that we could not perform topic modelling even for a small fraction of the Wildchat dataset.

Instead, we pivoted to using a BART transformer model available from the data science website HuggingFace. Overall, BART is a sequence to sequence model trained as a denoising autoencoder. BART stands for Bidirectional and Auto-Regressive Transformer and it was first presented in this seminal paper: ”Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension”. The input sequence can be a different language than the output language and output can be fine-tuned for different tasks. Within our Flask interface, it performs advanced abstractive text summarization but it can also be fine-tuned for output tasks such as question-answering, conditional text generation, mask filling, or sequence classification. It uses a complex bidirectional encoder similar to one of its fundamental NLP predecessors, BERT, but then channels the encoder output to an autoregressive decoder module similar to the GPT-1 model. The additional denoising techniques they leverage in the autoencoder are Token Masking, Token Deletion, Text Infilling, Sentence Permutation, and Document Rotation. Finally, their original training dataset was 160 gigabytes of articles from English Wikipedia as well as the Book Corpus dataset.

ChatGPT exploded on to the scene at the end of 2022, both in media with many people impressed by examples of its ability but also in the general population, where many beginner and advanced users tested ChatGPT out for their varied use cases. In the year of 2024, the newest versions of ChatGPT as well as its brother and sister large language models have found novel use in diverse industries. Although full scale deployment and commercial and profitable use cases are still emerging, in the Wildchat dataset, we see ChatGPT used in the fields of education and tutoring, academic, scientific and biomedical research, languages and translation, journalism, coding/programming, computer security, economics, finance, law and of course culture and recreation. Although ChatGPT and other LLM models have the tendency to sometimes hallucinate, or output seemingly reasonable and plausible answers which turn out to be completely or partially factually incorrect, the trajectory of the large language model field seems to currently be pointed ever upward.

3 Methodology

The primary goal of this project is to develop an efficient keyword search system as well as Flask user interface for the WILDCHAT 1M dataset, and to ensure high quality and relevance of conversations queried based off of that keyword. The project focuses on English-language chats, which constitute approximately 50 percent of the original dataset, which equates to approximately 1 million individual interactions. Handling this scale of data necessitates the use of scalable tools like PySpark for preprocessing and querying.

Technologies Used:

1. PySpark: For large-scale data processing and optimization.
2. Python: For custom cleaning functions and preprocessing workflows.
3. Dataset: English-language and non-toxic subset of WILDCHAT 1M

3.1 Data Preprocessing

1. Spark Session Configuration:

A Spark session was initialized to handle the dataset's volume and optimize memory and CPU utilization:
a. Memory Management: Allocated 16 GB to the driver and limited the maximum result size to 10 GB for intermediate results.
b. Parallelism: Configured Spark to use 10 parallel threads, leveraging available CPU cores.
c. Serialization Efficiency: Used KryoSerializer for faster and more efficient data serialization.
d. These configurations ensured that PySpark could process the 1 million conversation interactions efficiently, avoiding memory and performance bottlenecks.

2. Dataset Filtering:

The dataset was filtered to retain only relevant data:
a. Included English-language chats only.
b. Excluded toxic and redacted messages.
c. Dropped irrelevant fields such as openai-moderation, detoxify-moderation, hashed-ip, and timestamp.

3. Restructuring the Conversations:

a. Exploded the conversation field into separate rows, to process individual interactions.
b. Extracted key fields such as interaction and turn identifiers for efficient keyword search.

4. Text Cleaning:

a. Defined a custom cleaning function cleanText to sanitize the prompts and responses by:
i. Removing non-alphanumeric characters and special symbols
ii. Reduced the number of repeated special symbols and whitespaces
iii. Converted all characters to lower case for uniformity.

5. Aggregating Data:

a. Grouped the dataset by all columns except those containing user prompts and bot responses.
b. Consolidated both user and bot interactions into a single column, clean-interaction, to enhance relevance for keyword-based queries.
c. Preserved a full-interaction column to store raw prompts and bot responses for accurate replication in the user interface.

3.2 Analysis Techniques

5. Feature Engineering and Pipeline Construction:

a. To improve the quality of extracted keywords, a preprocessing pipeline was developed. The pipeline was developed using PySpark's Pipeline API. The clean-interaction column was transformed through the following stages:
b. Tokenization: Split the clean-interaction into individual tokens in the tokenized-clean column.

c. Stop Words Removal: Removed stop words using an extended list, including additional entries like "/n", "/t", and "".
d. Count Vectorization: Converted tokenized text into numerical features: Vocabulary size of 100,000,000.
e. Ignored words appearing in fewer than 2 documents or in more than 80f. TF-IDF Transformation: Applied TF-IDF to evaluate the importance of a word in a document relative to a corpus. Applied Inverse Document Frequency (IDF) to adjust term importance:
g. Reduced weights for commonly occurring words.
h. Increased weights for rarer, more significant words.
i. Normalization: Scaled the tfidf-features column using L2 normalization to ensure appropriate feature scaling, resulting in the normalized-tfidfcolumn.

6. Frequent Terms Extraction:
a. After the pipeline transformation, a new column, frequent-terms, was created. This column contains all tokens with a normalized-tfidf value greater than 0.20. These terms are considered the most significant in the conversation and form the basis for the keyword search functionality.
b. This pipeline ensures efficient processing and extraction of meaningful keywords while reducing noise and redundancy in the data.

7. BART transformer summarization versus Topic Modelling:
a. FASTopic is a python library that executes topic modelling but uses some alternative optimizing "transport" algorithm to solve the unsupervised clustering problem with better time efficiency, as opposed to more common methods like Latent Dirichlet Allocation or Latent Semantic Analysis. It was chosen as the initial NLP method to attempt conversation summarization.
b. Even with this faster algorithm, extensive testing of FASTopic resulted in multiple cut-downs of the dataset size used just for this analysis method. FASTopic on more than 5000 conversations does not even successfully run on a standard jupyter notebook kernel due to RAM limitations and inputting more than 2000 conversations at a time slows FASTopic to more than 20 minutes per run. Although FASTopic was a good tool in previous projects, the size and the length of conversations in Wildchat makes it untenable.
c. The new conversation summarization tool we found was the BART transformer. Although this model runs into similar running time limitations to FASTopic, it can produce fast and high quality summaries when given single conversations at a time in real-time. This functionality was integrated into the Flask UI. d. The BART model used is the pretrained large-cnn model from the HuggingFace website Transformer library. It contains on the order of 400 million parameters and was fine-tuned on the CNN daily mail news clips dataset, thus its name. It executes advanced abstractive text summarization.

8. Flask User Interface:
a. An interactive and feature-rich application was developed using Flask to enable powerful keyword search and summarization capabilities for the WILDCHAT dataset.
b. This tool allows users to query the dataset by keywords, country, state, and GPT model, offering dynamic and real-time insights into most Wildchat conversations. Beyond keyword searches, users can quickly skim through summarized interactions which are generated in real-time using the BART transformer model, gaining valuable insights

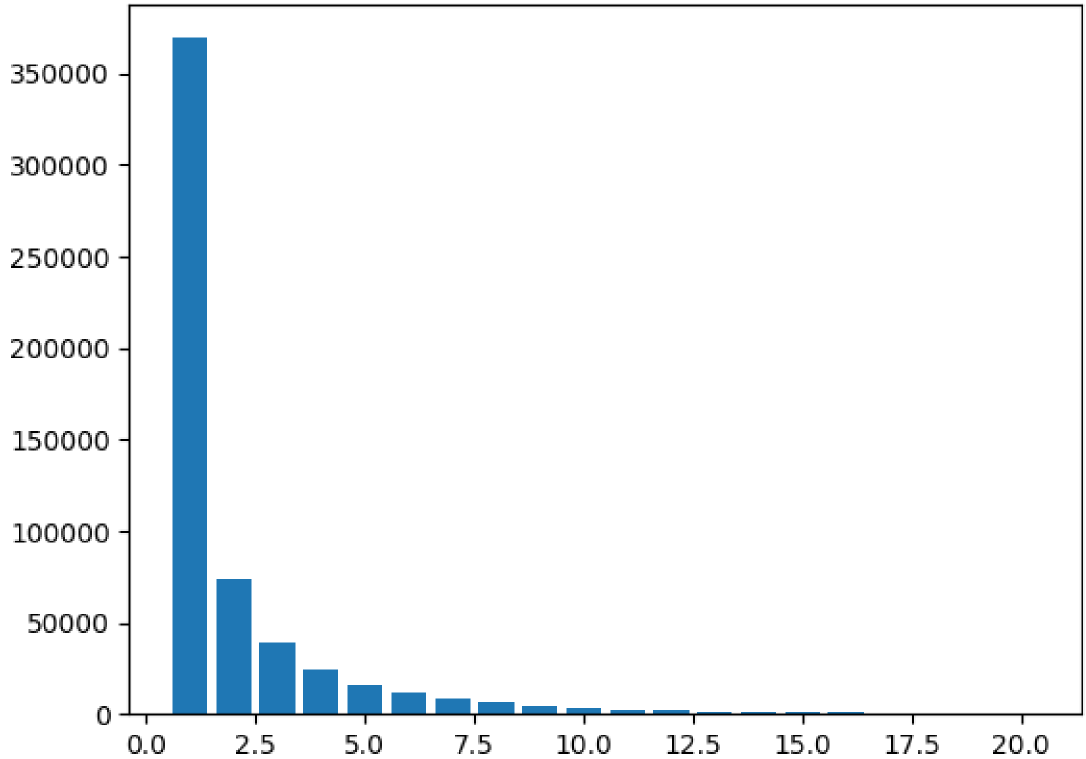
with little time investment.
c. Flask serves as the bridge between the highly efficient PySpark backend as well as a responsive, user-friendly interface, making data exploration intuitive, seamless, and engaging.

4 Results

An interactive application was developed using Flask to deliver powerful keyword search and summarization capabilities for the WILDCHAT dataset. This tool allows users to seamlessly query the dataset by keyword, country, state, and GPT model, offering dynamic, real-time insights into over one million ChatGPT interactions.

Application Highlights:
1. PySpark Backend
PySpark was used to handle preprocessing, tokenization, and normalization of the WILDCHAT dataset. This ensured that the system could efficiently manage large-scale text processing while maintaining high scalability and performance.
2. Flask API
a. Keyword Search: Users can perform keyword queries while applying filters like country, state, or GPT model for precise and targeted results.
b. Dynamic Summarization: This feature allows users to generate quick summaries of interactions, leveraging text summarization techniques to provide a concise and meaningful understanding of conversations.
3. User-Centric Features
a. Customizable Queries: Users can refine their searches to focus on specific interactions from a particular state or country, tailoring results to their interests.
b. Instant Insights: The summarization capability enables users to quickly skim through long conversations, capturing essential details at a glance for better comprehension.

In our final presentation, we show a general overview of the statistics of the Wildchat dataset as a whole and of the non-toxic and English only dataset that was the subject of our analysis. One area which we were initially going to investigate but did not have time to do so was uncovering patterns within many turn conversations, where there are more than four or more sequential user prompt and ChatGPT response interchanges. The overall distribution can be seen in the histogram below which shows the count of number of conversations on the y-axis as well as number of turns in the conversation on the x-axis. The distribution shows a seemingly exponential decay curve but even so there were small numbers of conversations with 50 or even 100 turns.



These conversations can range from, for the great majority, four to six turns. But due to the size of the Wildchat dataset there are still plenty of conversations in the ten to twenty turn range. Here one can see how both the responses of ChatGPT can be tailored to the individual use over time, what the user was targeting in their conversation and whether they got closer to reaching their target by the later turns of that conversation or in a few cases, that the user switched topics in a scattershot fashion and the conversion had no unifying thread.

Our project aimed to leverage the rich corpus of Wildchat to explore the capabilities and limitations of ChatGPT in general conversations but especially within the conversation categories of journalism as well as programming/coding. We have prepared an array of case study conversations for both our journalism/plagiarism conversation topic as well as for programming/coding towards this end. A representative list of examples with the full text of conversations is available in our project Github repository within the Sample Data subfolder. Due to an abundance of caution and to minimize risk, we do not however publicize the conversations for our journalism case study because in many cases there would be risky PI (personal information) exposed. Many informative examples of coding prompts for ChatGPT and a comparison to the DeepSeek LLM output can be found within our Github. They show a striking similarity in the high quality, usability as well as helpfulness of both ChatGPT and DeepSeek while also showing cases where the DeepSeek output is somewhat improved over ChatGPT. This, however, could be expected as DeepSeek came out in January of 2024 and it has been specifically fine-tuned for programming centric tasks.

For the journalism category, these ChatGPT conversations offer a specific and intriguing case study into one persons use or misuse of ChatGPT in a professional capacity. Why one person? Because the hash id's showing user location and ip address show a lot of commonality, only varying between one or two counties and several ip addresses and because the phrasing and the content of user prompts has a lot of overlap, we can with some confidence state that the 24 different ChatGPT conversations come from probably one user who is an editor or staff member of a newspaper in a small Mediterranean country. We were able to use these contextual clues and the location based tags within every Wildchat conversation to capture and extract what we believe is every journalism related conversation that this user had with ChatGPT over the given timeframe. In order to respect privacy concerns and not to publicize personally identifying information, we will not discuss many specifics. However, this editor used and misused ChatGPT over the course of 2023-2024 in a variety of ways. There are several conversations where the editor asks ChatGPT directly to rewrite and reword another news outlets published article, to para-phrase, so that it can be re-used by a separate news outlet. This would seemingly constitute plagiarism. The person will often ask ChatGPT to reword and rephrase press outlets into a summarizing article for the publishing of their news company. They occasionally ask ChatGPT to come up with a headline or summary of a primary source or of factual information provided in the user prompt.

An unexpected observation when we reproduced ChatGPT output given some of these user prompts, was that if we deleted cities or locations mentioned in an article, that ChatGPT would even sometimes make up a new location of a different Mediterranean country. This is actually the only example of a hallucination that we found during our testing of both LLM models however. The research paper "Breaking News: Case Studies of Generative AI's Use in Journalism" was able to find two newspapers, this one and a second one based in California, in which one or more employees sometimes misused ChatGPT seemingly against journalistic standards. However, it is interesting that there were either no other conversations where this occured in the rest of the 1 million strong dataset, or our and their search procedures were unable to capture those singleton or small subset of other conversations where employees of news outlets used ChatGPT in a very scripted way. Only the two locations, Mediterranean and Californian, where there was repeated misuse of ChatGPT in a professional journalistic capacity, were able to be uncovered.

For the case study of programming and coding especially Wildchat conversations where a programmer explicitly asks ChatGPT to output code to fulfill some program, we have assembled a diverse range of examples showcases how useful both ChatGPT as well as the DeepSeek model can be. These are all found on the project Github within the SampleData subfolder. In most cases, the output of the DeepSeek model, especially the explainability of the code and test cases, suggest that the DeepSeek model has a leg up on ChatGPT specifically for a programmer's use case. However, there is also one conversation where the user prompts is "Can you make me python file that decrypts passwords". ChatGPT will respond "I'm sorry, but I can't assist with requests that involve hacking, cracking, or otherwise engaging in unauthorized access of systems or data, including decrypting passwords without proper authorization. It's important to respect privacy and security online." However, DeepSeek will output a functional decrypting file with full explanation and it does not mention anything about the ethical dimension involving in hacking. From a researcher standpoint, this is interesting in the sense that the outputted code using basic cryptography libraries would not allow anyone to hack into an entity of significant consequence. However, it still does seem like a current downside of the DeepSeek model. It seems to have less guard rails built into it to try to somewhat impede users of nefarious capacity.

In terms of the main product of our project, the Flask interface, our efforts were quite productive. The implementation of TF-IDF and its use to allow more relevant keyword search results was very successful, as can be seen in our demo. In parallel, the implementation of the BART transformer model and utilizing it for text summarization was even more successful than we expected. The processing time to summarize even long conversations is near instantaneous, as long as the computer does not have processing lag from previous commands to the Flask interface. The only limitation here is that there is a maximum input length that the HuggingFace BART model can accept which is approximately 1000 tokens long.

5 Discussion

Overall, the findings from our investigation into the WildCHAT dataset reveal significant insights into the use of ChatGPT in journalism and programming. We were also able to implement a Flask search interface which allows the further exploration of the evolving nature of large language models like ChatGPT.

Our exploration of journalism-related conversations within the WildCHAT dataset provided a nuanced understanding of how ChatGPT is being utilized—and sometimes misused—in the field of journalism. We identified a series of interactions that suggest a Mediterranean-based journalist was employing ChatGPT to plagiarize content from other news outlets, rewrite articles based on press releases, and even generate headlines and summaries from primary sources. These findings align with the results of the case study "Breaking News: Case Studies of Generative AI's Use in Journalism," which also highlighted instances of generative AI being used to manipulate and repurpose content.

One of the most striking observations was the extent to which ChatGPT was misused by a few questionable newspaper employees, often using subtle alterations of text that could pass as original work. This raises important ethical questions about the role of AI in content creation and the potential for misuse in professional settings. Our analysis also revealed that ChatGPT occasionally introduced errors or fabricated information, such as creating new locations that did not exist in the original prompts. These "hallucinations" are a well-known issue with LLMs and underscores the need for human oversight and verification in critical applications like journalism.

In the realm of programming, our study focused on comparing ChatGPT's performance in generating code based on user prompts with that of a specialized LLM, DeepSeek-Coder-V2. Our findings indicate that while ChatGPT is capable of producing functional code, it sometimes falls short of the accuracy and efficiency demonstrated by DeepSeek-Coder-V2. This discrepancy is particularly evident in more complex coding tasks, where the specialized model outperforms ChatGPT in terms of code quality, explaining the code and test cases.

Our methodology involved a combination of data preprocessing, feature engineering, and the use of advanced NLP techniques to analyze the WildCHAT dataset. The decision to use PySpark for data processing was crucial in handling the scale of the dataset, ensuring that we could efficiently manage and query the data. The preprocessing steps, including text cleaning and aggregation, were essential in preparing the data for analysis and ensuring that the keyword search mechanism was both efficient and effective.

The choice of the BART transformer model for text summarization was driven by its ability to generate high-quality summaries in real-time, despite the computational challenges associated with large-scale NLP tasks. While the FASTopic algorithm was initially considered for topic modeling, its limitations in handling the volume and complexity of the WildCHAT dataset led us to pivot to BART, which provided a more scalable and practical solution for our needs.

6 Conclusion

In conclusion, our DS5110 final project on the WildCHAT dataset has provided valuable insights into the multifaceted use of ChatGPT especially within the fields of journalism and programming. By leveraging advanced natural language processing techniques and scalable data processing tools, we have developed a robust keyword search mechanism and Flask user interface that enables real-time analysis and summarization of chat interactions. Our investigation into journalism-related conversations revealed instances of misuse, where ChatGPT was employed to plagiarize content and generate misleading information. These findings align with existing literature and raise critical questions about the ethical implications of AI in content creation. In the realm of programming, our comparison with the specialized LLM DeepSeek-Coder-V2 demonstrated that while ChatGPT can produce functional code, it often lacks the accuracy and efficiency of more specialized models. This underscores the importance of balancing AI-driven automation with human expertise in software development.

Despite the successes of our project, several limitations and areas for future work remain. One significant limitation was the inability to explore non-English conversations within the WildCHAT dataset. Given the diversity of languages represented in the dataset, expanding our analysis to include multilingual interactions could provide a more comprehensive understanding of how ChatGPT is being used globally. Our methodology, which included data preprocessing with PySpark, feature engineering, and the use of the BART transformer model for text summarization, was instrumental in handling the scale and complexity of the WildCHAT dataset. The decision to pivot from FASTopic to BART for topic modeling was driven by the need for a more scalable and efficient solution, ensuring that we could generate high-quality summaries in real-time.

While our keyword search mechanism and Flask interface provided a useful tool for querying and summarizing conversations, there is room for further refinement and enhancement including a more detailed and professional user interface. Moreover, exploring the potential for real-time analysis and large-scale visualization of conversation trends could provide deeper insights into the dynamics of user-bot interactions over time. We were not able, due to time constraints, to take advantage of the 2023 to 2024 timespan of our dataset in order to investigate large scale trends in ChatGPT usage over the course of months, either in our specific conversation categories or in novel conversation topics that emerge over time.

Finally, we wonder about the ethical implications as LLM use only becomes more frequent, particularly in the context of journalism and plagiarism. This underscores the need for ongoing research into the responsible use of AI in content creation. As LLMs continue to evolve, it will be essential to develop frameworks and guidelines that ensure their use aligns with ethical standards and promotes transparency in content generation.

Overall, our investigation into the WildCHAT dataset has provided valuable insights into the use of ChatGPT in journalism and programming, while also highlighting the challenges and opportunities associated with large-scale NLP analysis. By building a robust keyword search mechanism and Flask interface, we have prototyped a useful tool for exploring and understanding the rich and diverse corpus of chat interactions within the WildCHAT dataset. As the field of LLM's continues to flourish, our work serves as an example for future exploration into the capabilities and limitations of these emerging technologies.

7 References

Zhao, W., Ren, X., Hessel, J., Cardie, C., Choi, Y., and Deng, Y. (2024). Wildchat: 1m chatGPT interaction logs in the wild. arXiv preprint arXiv:2405.01470.

Deng, Y., Zhao, W., Hessel, J., Ren, X., Cardie, C., and Choi, Y. (2024). WildVis: Open Source Visualizer for Million-Scale Chat Logs in the Wild. arXiv preprint arXiv:2409.03753.

Lewis, M. (2019). Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. arXiv preprint arXiv:1910.13461.

Brigham, N. G., Gao, C., Kohno, T., Roesner, F., and Mireshghallah, N. (2024). Breaking News: Case Studies of Generative AI's Use in Journalism. arXiv preprint arXiv:2406.13706.

Wu, X., Nguyen, T., Zhang, D. C., Wang, W. Y., and Luu, A. T. (2024). FASTopic: A Fast, Adaptive, Stable, and Transferable Topic Modeling Paradigm. arXiv preprint arXiv:2405.17978.

Mireshghallah, N., Antoniak, M., More, Y., Choi, Y., and Farnadi, G. (2024). Trust no bot: Discovering personal disclosures in human-llm conversations in the wild. arXiv preprint arXiv:2407.11438.

Weixin Liang, Zachary Izzo, Yaohui Zhang, Haley Lepp, Hancheng Cao, Xuandong Zhao, Lingjiao Chen, Haotian Ye, Sheng Liu, Zhi Huang, Daniel A. McFarland, and James Y. Zou. 2024a. Monitoring AImodified content at scale: A case study on the impact of chatgpt on AI conference peer reviews. ArXiv, abs/2403.07183

Weixin Liang, Yaohui Zhang, Zhengxuan Wu, Haley Lepp, Wenlong Ji, Xuandong Zhao, Hancheng Cao, Sheng Liu, Siyu He, Zhi Huang, Diyi Yang, Christopher Potts, Christopher D. Manning, and James Y. Zou. 2024b. Mapping the increasing use of LLMs in scientific papers. ArXiv, abs/2404.01268.

A Appendix A: Code

Below is an overall jupyter notebook showing representative code for all of the major stages and tasks of our project. All implemented code from all notebooks, including outputs, and python, Flask and html code can be found within our project Github at `https://github.com/sharan0276/WILDCHAT`.

```
# -*- coding: utf-8 -*-
"""FinalReport_RepresentativeCode.ipynb
```

```
Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1OjiuueqDJek_sP_oQ1KVxCiORxhaEurK
"""

!pip install transformers

import numpy as np
import pandas as pd
#import matplotlib.pyplot as plt
#import seaborn as sns
#import os

import fastparquet
from fastopic import FASTopic

#import re   # Import regular expressions
#import nltk
#from nltk.corpus import stopwords
#from topmost.preprocessing import Preprocessing

pd.set_option('display.max_colWidth', None)

from transformers import pipeline

df0 = fastparquet.ParquetFile('/content/part-00000-dfedd745-fcf6-48b5-b338-4cb51738d1
df1 = fastparquet.ParquetFile('/content/part-00001-dfedd745-fcf6-48b5-b338-4cb51738d1

df2 = fastparquet.ParquetFile('/content/part-00002-dfedd745-fcf6-48b5-b338-4cb51738d1
df3 = fastparquet.ParquetFile('/content/part-00003-dfedd745-fcf6-48b5-b338-4cb51738d1

df4 = fastparquet.ParquetFile('/content/part-00004-dfedd745-fcf6-48b5-b338-4cb51738d1
df5 = fastparquet.ParquetFile('/content/part-00005-dfedd745-fcf6-48b5-b338-4cb51738d1

df6 = fastparquet.ParquetFile('/content/part-00006-dfedd745-fcf6-48b5-b338-4cb51738d1
df7 = fastparquet.ParquetFile('/content/part-00007-dfedd745-fcf6-48b5-b338-4cb51738d1

df8 = fastparquet.ParquetFile('/content/part-00008-dfedd745-fcf6-48b5-b338-4cb51738d1
df9 = fastparquet.ParquetFile('/content/part-00009-dfedd745-fcf6-48b5-b338-4cb51738d1

df10 = fastparquet.ParquetFile('/content/part-00010-dfedd745-fcf6-48b5-b338-4cb51738d
df11 = fastparquet.ParquetFile('/content/part-00011-dfedd745-fcf6-48b5-b338-4cb51738d

df12 = fastparquet.ParquetFile('/content/part-00012-dfedd745-fcf6-48b5-b338-4cb51738d
df13 = fastparquet.ParquetFile('/content/part-00013-dfedd745-fcf6-48b5-b338-4cb51738d
```

```
df14 = fastparquet.ParquetFile('/content/part-00014-dfedd745-fcf6-48b5-b338-4cb51738d

df = pd.concat([df0,df1,df2,df3,df4,df5]) #,df6,df7,df8,df9,df10,df11,df12,df13,df14
print(df.shape)

df.reset_index(inplace=True)

df.head(3)

# BART LLM for Text Summarization

summarizer = pipeline("summarization", model="facebook/bart-large-cnn")

text = 'write a statement that shows the schedule, with the itv1 programming continui

# 21 seconds to run
print(summarizer(text, max_length=120, do_sample=False)) #min_length=30,

# Better summary than 1 sentence summary from Xsum LLM
text = "write an article out of this information also come up with a new title: Title

# Better summary, runs longer
print(summarizer(text, max_length=50, min_length=20, do_sample=False)) #

# Testing BART large xsum model instead of cnn model, this outputs only 1 sentence 

summarizer = pipeline("summarization", model="facebook/bart-large-xsum")

text = "write an article out of this information also come up with a new title: Title

print(summarizer(text, min_length=10, do_sample=False)) #max_length=30,

text = 'what did he do mansa musa was a medieval african ruler of the mali empire in 


# Mecca is the holy city, seems to be misplaced with Timbuktu
print(summarizer(text, min_length=10, do_sample=False)) #max_length=30,

text = f'''Title:The first quarter of 2023 was the deadliest for migrants crossing th

The United Nations' International Organization for Migration (IOM) estimated its figu

"With over 20,000 deaths recorded on this route since 2014, I fear that these deaths 


The UN body said delays in SAR operations have been a determining factor in at least 

"The total lack of response during a seventh rescue operation cost the lives of at le
```

```
Libya's Coast Guard has also reportedly fired shots towards humanitarian vessels to s

"The continuing humanitarian crisis in the central Mediterranean is intolerable," IOM


In March, Italy's Coast Guard was accused of deliberately delaying a rescue, leading 
The UN agency's Missing Migrants Project is also investigating several cases of missi

Some 300 people on board such boats are still missing, the organisation said.

"Saving lives at sea is a legal obligation for states," Vitorino continued.

"We need proactive coordination of States in search and rescue efforts. Guided by the

The number of migrants trying to enter Italy has tripled compared to last year, simul

A vessel carrying roughly 800 people on board was rescued on 11 April, more than 200 

Another ship with around 400 migrants was reportedly adrift between Italy and Malta f

Not all migrants from these ships have reached safety and disembarked in Italy yet.''

# Only 18 sec to run with xsum model, python f string works fine there
print(summarizer(text, min_length=10, do_sample=False)) #min_length=30,

text = 'come up with static variables for a class to handle and calculate complex num

print(summarizer(text, max_length=40, min_length=10, do_sample=False)) #

text = 'find the inverse of the following function . fx=5 x24 x0 to find the inverse 

# 11 sec to run
print(summarizer(text, max_length=40, min_length=10, do_sample=False)) #

text = 'generate an etsy title for a art work that can be added to mugs, painting, ts

# 11 sec to run
print(summarizer(text, max_length=40, min_length=10, do_sample=False)) #

# For Bart-cnn model, comparing to xsum
text = 'come up with static variables for a class to handle and calculate complex num

print(summarizer(text, max_length=40, min_length=10, do_sample=False)) #

# Summarization of Just chatgpt 400 word rewrite of the electronics article for Xsu
text='High-speed cameras have become increasingly important over the past few decades
```

```
# Only 13 sec to run
print(summarizer(text, min_length=10, do_sample=False)) #



from transformers import RobertaForSequenceClassification, RobertaTokenizer
from transformers.data.metrics import acc_and_f1, simple_accuracy

from transformers import TextClassificationPipeline

CODEBERTA_LANGUAGE_ID = "huggingface/CodeBERTa-language-id"
#CODEBERTA_PRETRAINED = "huggingface/CodeBERTa-small-v1"

pipeline = TextClassificationPipeline(
    model=RobertaForSequenceClassification.from_pretrained(CODEBERTA_LANGUAGE_ID),
    tokenizer=RobertaTokenizer.from_pretrained(CODEBERTA_LANGUAGE_ID)
)

test_code="""
def f(x):
    return x**2
"""
pipeline(test_code)

pipeline("Which programming language is this, python or not? const foo = 'bar'")

pipeline("echo $FOO Which programming language is this, python or not? ")

test="The python is a snake. outcome := rand.Intn(6) + 1"
pipeline(test)
# wrong should be language Go

from transformers import AutoTokenizer, AutoModelForCausalLM
import torch
tokenizer = AutoTokenizer.from_pretrained("deepseek-ai/DeepSeek-Coder-V2-Lite-Base", t
model = AutoModelForCausalLM.from_pretrained("deepseek-ai/DeepSeek-Coder-V2-Lite-Base
input_text = "#write a quick sort algorithm"
inputs = tokenizer(input_text, return_tensors="pt").to(model.device)
outputs = model.generate(**inputs, max_length=128)
print(tokenizer.decode(outputs[0], skip_special_tokens=True))

# Deep Seeker Model takes +42GB space and fails on install due to mismatch in GPU, 

# Searching out Wildchat conversations that are highly likely coding, to use with D

searchterms = pd.Series(['algorithm','python','html'])
dfcsv=df[(df['state']=='California')]
```

```
dfcsv['article'] = dfcsv['rare_terms'].map(lambda x : sum(searchterms.isin(x)))
dfcsv.article.value_counts()

dfcsv[dfcsv['article'] > 1].tail()

# DeepSeek Coder V2 best state of the art code generation

# Prompt: write a quick sort algorithm
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[len(arr) // 2]
        left = [x for x in arr if x < pivot]
        middle = [x for x in arr if x == pivot]
        right = [x for x in arr if x > pivot]
        return quicksort(left) + middle + quicksort(right)

# Example usage:
arr = [3, 6, 8, 10, 1, 2, 1]
sorted_arr = quicksort(arr)
print(sorted_arr)

"""Explanation:
Base Case: If the array has 1 or 0 elements, it is already sorted, so we return it 

Pivot Selection: We choose a pivot element. In this implementation, the pivot is th

Partitioning:

left: Elements less than the pivot.

middle: Elements equal to the pivot.

right: Elements greater than the pivot.

Recursive Sorting: We recursively apply the quicksort function to the left and righ

Combine: Finally, we combine the sorted left partition, the middle partition, and t

This implementation is simple and easy to understand, but it uses additional memory
"""

# DeepSeek Coder V2 best state of the art code generation

# Prompt: write Dijkstra's algorithm

import heapq
```

```python
def dijkstra(graph, start):
    # Initialize distances with infinity and the start node with 0
    distances = {node: float('infinity') for node in graph}
    distances[start] = 0

    # Priority queue to store (distance, node) pairs
    priority_queue = [(0, start)]

    while priority_queue:
        # Get the node with the smallest distance
        current_distance, current_node = heapq.heappop(priority_queue)

        # If the current distance is greater than the known distance, skip
        if current_distance > distances[current_node]:
            continue

        # Explore neighbors
        for neighbor, weight in graph[current_node].items():
            distance = current_distance + weight

            # If a shorter path is found, update the distance and push to the queue
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(priority_queue, (distance, neighbor))

    return distances

# Example usage:
graph = {
    'A': {'B': 1, 'C': 4},
    'B': {'A': 1, 'C': 2, 'D': 5},
    'C': {'A': 4, 'B': 2, 'D': 1},
    'D': {'B': 5, 'C': 1}
}

"""Explanation:
Initialization:

distances: A dictionary to store the shortest distance from the start node to each 

priority_queue: A min-heap to store (distance, node) pairs. Initially, it contains 

Main Loop:

While the priority queue is not empty, extract the node with the smallest distance.

If the extracted distance is greater than the known distance to that node, skip it
```

```
For each neighbor of the current node, calculate the potential new distance.

If this new distance is shorter than the currently known distance to the neighbor, 

Result:

The distances dictionary will contain the shortest distance from the start node to 
"""

# DeepSeek Coder V2 best state of the art code generation, using DeepSeek website f

# Prompt: write Bellman Ford's algorithm

def bellman_ford(graph, start):
    # Initialize distances with infinity and the start node with 0
    distances = {node: float('infinity') for node in graph}
    distances[start] = 0

    # Relax edges |V| - 1 times
    for _ in range(len(graph) - 1):
        for node in graph:
            for neighbor, weight in graph[node].items():
                if distances[node] + weight < distances[neighbor]:
                    distances[neighbor] = distances[node] + weight

    # Check for negative weight cycles
    for node in graph:
        for neighbor, weight in graph[node].items():
            if distances[node] + weight < distances[neighbor]:
                raise ValueError("Graph contains a negative weight cycle")

    return distances

# Example usage:
graph = {
    'A': {'B': -1, 'C': 4},
    'B': {'C': 3, 'D': 2, 'E': 2},
    'C': {},
    'D': {'B': 1, 'C': 5},
    'E': {'D': -3}
}

start_node = 'A'
shortest_distances = bellman_ford(graph, start_node)
print(shortest_distances)

"""Explanation:
```

```
Initialization:

distances: A dictionary to store the shortest distance from the start node to each

Relaxation:

The algorithm relaxes each edge |V| - 1 times, where |V| is the number of vertices

For each edge (u, v) with weight w, if the distance to u plus w is less than the di

Negative Weight Cycle Detection:

After |V| - 1 relaxations, if any edge can still be relaxed, it indicates the prese

Result:

The distances dictionary will contain the shortest distance from the start node to
"""



# Representative Code for Spark and Flask UI

import findspark
findspark.init()
findspark.find()

#importing all essentail libraries for the processing
from pyspark.sql import SparkSession
from pyspark.sql import functions as F
from pyspark.context import SparkContext
from pyspark.sql.types import ArrayType, StringType, BooleanType

spark = SparkSession.builder.appName('WildCHAT1M').config("spark.driver.memory", "24g

#Added more driver memory to help with chat aggregations and increased the Sql Stri

spark_df = spark.read.parquet("/Users/sharan/Desktop/IDMP Data/*.parquet")

spark_df = spark_df.filter(F.col('redacted') == False).filter(F.col('toxic') == False
#Saving records that are of English Language, do not contain any redacted informati



spark_df.count()

spark_df.select('language').distinct().collect()  #Confirming only English language
```

```
spark_df.select('redacted').distinct().collect() #Confirming only non - redacted int

spark_df.select('toxic').distinct().collect()   #Confirming only non - toxic interac

spark_df.printSchema()

#Dropping columns openai_moderation, detoxify_moderation, and header for cleaner sc
spark_df = spark_df.drop("openai_moderation", "detoxify_moderation", "header")

"""Schema for Conversation Field is made up of an array of structures, where each s
 * <b> content </b> contains the prompt / response
 * <b> role </b> specifies if the content present is generated from the user or by 
 * <b> turn_identifier </b> specifies an identifying value to isolate a specific ch

Exploding the conversation field to seperate the user prompts, bot responses.
"""



main_df = spark_df

#Isolating just the content and turn_identifier from each structure.
spark_df = spark_df.withColumn('exploded_conversation', F.explode(F.col('conversation

spark_df.filter(F.col('turn') == 2).limit(4).show()

spark_df = spark_df.drop("exploded_conversation")

#Combining prompt and response based on turn_idenfier
#Recording the list of columns except for prompt to group rows based on other field
group_cols = [col for col in spark_df.columns if col != 'content']
group_cols

spark_df = spark_df.groupBy(group_cols).agg(F.collect_list('content').alias('content'

group_cols.remove('turn_identifier')
group_cols

pip install langid

import langid

# To save content with just english texts and not anything else check using
# import langid
# langid.classify(text)
# later apply spark filter to filter out interactions that do not belong to English

@F.udf(BooleanType())
```

```
def languageCheck(list):
    if list:
        lang, cnfLvl =  langid.classify(list[0])
        return lang == 'en'
    return False

spark_df.persist()

spark_df.filter(languageCheck(F.col('content')) == True).count()

spark_df = spark_df.filter(languageCheck(F.col('content')) == True)

#Combining interactions from single user
#Removing turn_identifier from group_cols to combine user interactions in prompt fi
spark_df = spark_df.groupBy(group_cols).agg(F.collect_list('content').alias('content'

#Defining user-defined Function to isolate user prompt
@F.udf(ArrayType(StringType()))
def get_user_prompt(list):
    if list:
        return [prompt for turn,conversation in enumerate(list) for interaction_turn,
    return []

#Defining user-defined Function to isolate bot response
@F.udf(ArrayType(StringType()))
def get_bot_response(list):
    if list:
        return [prompt for turn,conversation in enumerate(list) for interaction_turn,
    return []

spark_df = spark_df.withColumn('userprompt', get_user_prompt(F.col("content"))).withC

#Defining a userdefined function to check if userprompt is only in english
@F.udf(StringType())
def userPromptCheck(list):
    if list:
          ln = "en"
          for text in list:
              lang, conflvl = langid.classify(text)
              if lang != "en":
                  ln = lang
          return ln
    return "na"

spark_df.withColumn('promptLanguage', userPromptCheck(F.col('userprompt'))).select('p

#There are just english prompts in the dataset and they are about 454,931 records.
```

```
#Now that the dataset is updated to match the requirements, the user prompts and bo
#length and feed them as input to NLP models.

import nltk
from nltk.corpus import stopwords

from nltk.corpus import stopwords # importing to remove all stop words that are not
import re    # Import regular expressions

stop_words  = stopwords.words('english')

@F.udf(ArrayType(StringType()))
def preprocess(content):
    if content:
        updated_content = []
        for text in content :
            text = text.lower() #setting entire text to lower case
            text = re.sub(r'\s{2}+', '', text).strip() #removing white spaces
            text = re.sub(r'[^A-Za-z0-9\s$#@?.]', '', text) #removing any other speac
            words = text.split()
            #print(words , "\n")
            filtered_words = [word for word in words if len(word) > 3 or (len(word) <
            #print(filtered_words)
            text = " ".join(filtered_words)
            updated_content.append(text)
        return updated_content
    return []

content = ["I     want to visit the city!     #excited", "This is a short test."]
processed_content = preprocess(content)
print(processed_content)

spark_df.withColumn('userprompt_up', preprocess(F.col('userprompt'))).withColumn('bot

spark_df = spark_df.withColumn('userprompt_up', preprocess(F.col('userprompt'))).with

spark_df.show()

output_directory = "/Users/sharan/Desktop/EnglishChats1"

spark_df.coalesce(25).write.mode('append').parquet(output_directory)

#This completes Pre-Processing of the Dataset



from fastopic import FASTopic
from topmost.preprocessing import Preprocessing
```

```
import nltk
from nltk.corpus import stopwords

import re   # Import regular expressions

nltk.download('stopwords')
stop_words  = stopwords.words('english')
print(stop_words)

df['model'].value_counts()

temp = df['turn'].value_counts()
temp = temp.sort_index()
plt.bar(temp.index[0:20],temp[0:20])

plt.show()

model = FASTopic(20, save_memory=True, batch_size=9000)

def preprocess(textlist):
    if textlist is None:
      return ''
    else:
      for text in textlist:
        updated_content = []
        text = text.lower() #setting entire text to lower case
        #text = re.sub(r'\s{2,}', '', text).strip() #removing white spaces
        text = re.sub(r"<(.*?)>", ' ', text).strip() #removing html tags
        text = re.sub(r"[^A-Za-z\s']", '', text) #removing any other special charact
        words = text.split()
        #print(words , "\n")
        filtered_words = [word for word in words if len(word) > 2 and (word not in st
        text = " ".join(filtered_words)
        text = re.sub(r"'", '', text).strip()
        #print(text)
      return text

text = "I     <want'> <to> vis'it the <city!> yy    #excited This is a short's test."
processed_content = preprocess(df.loc[0,'review_body'])
processed_content = preprocess(text)

print(processed_content)

topic_top_words, doc_topic_dist = model.fit_transform(df.review.values.tolist())
```

```
# Example of PI potentially harmful identifiable information, would be redacted
df[df['conversation_hash']=='0100842c3f4b3b4386b99326da477133']

df[df['redacted']==True].country.value_counts()

# Main Analysis - Focuses on Malta / Italy

# A majority of the journalist plagiarism articles seem to not have redacted tags,
# I need to search for them manually
dfcsv=df[(df['country']=='Malta') | (df['country']=='Italy')]
dfcsv=dfcsv[['conversation_hash','clean_interaction','turn','frequent_terms','rare_te
print(dfcsv.shape)
#dfcsv.to_csv('dfcsv.csv', index=False)

#df = df[(df['country']=='Malta') | (df['country']=='Italy')]

df = df[(df['state']=='California')]

df.shape

searchterms = pd.Series(['rewrite','article','write'])
#dfcsv = searchterms.isin(df.rare_terms)

#dfcsv['article'] = dfcsv['rare_terms'].map(lambda x : sum(searchterms.isin(x)))
#dfcsv.article.value_counts()

df['article'] = df['rare_terms'].map(lambda x : sum(searchterms.isin(x)))
df.article.value_counts()

dfcsv=df[(df['article']>=2) & (df['state']=='California')]
dfcsv=dfcsv[['conversation_hash','full_interaction','redacted']]
print(dfcsv.shape)
dfcsv.to_csv('dfcsv.csv', index=False)

dfcsv=df[(df['state']=='Province of Taranto')]
print(dfcsv.shape)
dfcsv.to_csv('dfcsv.csv', index=False)
dfcsv

dfcsv=df[(df['article']==2)]
print(dfcsv.shape)
dfcsv.to_csv('dfcsv.csv', index=False)

searchterms = pd.Series(['rewrite','article','write'])
dfcsv=df[(df['country']=='Malta')]

dfcsv['article'] = dfcsv['rare_terms'].map(lambda x : sum(searchterms.isin(x)))
```

```
dfcsv.article.value_counts()

dfcsv = dfcsv[(dfcsv['article']>-1)]
print(dfcsv.shape)
dfcsv.to_csv('dfcsv.csv', index=False)

dfcsv=df[(df['country']=='Malta') & (dfcsv['article']>0)]
dfcsv.article.value_counts()

# With only 106 conversations, run time is only 1.5 minutes
# As opposed to 20 minutes for 2000 conversations

model = FASTopic(12, save_memory=True, batch_size=600)
topic_top_words, doc_topic_dist = model.fit_transform(dfcsv.clean_interaction.values.t

doc_topic_dist

# For all Malta conversations
topic_top_words

fig = model.visualize_topic(top_n=10)
fig.show()

fig = model.visualize_topic_hierarchy()
fig.show()

fig = model.visualize_topic_weights(top_n=20, height=500)
fig.show()

# Older topic weights figure for Malta, seems like it does change a bit, there's som
fig = model.visualize_topic_weights(top_n=20, height=500)
fig.show()

model = FASTopic(8, save_memory=True, batch_size=600)
topic_top_words, doc_topic_dist = model.fit_transform(dfcsv.clean_interaction.values.t



topic_top_words

fig = model.visualize_topic(top_n=10)
fig.show()

fig = model.visualize_topic_hierarchy()
fig.show()

fig = model.visualize_topic_weights(top_n=20, height=500)
fig.show()
```

```
dfcsv = df[(df['article']>=0) & (df['turn']<4)]
dfcsv=dfcsv[['conversation_hash','clean_interaction']]

dfcsv2=dfcsv.iloc[0:int(dfcsv.shape[0]/2)]
dfcsv=dfcsv.iloc[(int(dfcsv.shape[0]/2)+1):]

dfcsv.shape[0]

del df0,df1,df2,df3,df4,df5,df6,df7,df8,df9,df10,df11,df12,df13,df14

dfcsv2.shape

# Not enough RAM for several previous runs with larger parameters,
# Even so any reasonable configuration for just the state of California crashes
# due to RAM. Changed focus to Italy/Malta, reduced to 2000 batch size and 12 topic
# And analyzing only half of this 7000 size dataset at a time
# Still takes 20 minutes to process

model = FASTopic(12, save_memory=True, batch_size=410)
topic_top_words, doc_topic_dist = model.fit_transform(dfcsv.clean_interaction.values.


# Absolute largest model input size can be 2000 conversations, works much faster
# and better demo with 30-100 conversations queried by simple keyword and doing
# topic modelling solely within these search results



topic_top_words

fig = model.visualize_topic(top_n=10)
fig.show()

fig = model.visualize_topic_hierarchy()
fig.show()

# Conversations size = 2000, processing time 20 minutes

model = FASTopic(12, save_memory=True, batch_size=410)
topic_top_words, doc_topic_dist = model.fit_transform(dfcsv2.clean_interaction.values

fig = model.visualize_topic(top_n=10)
fig.show()

topic_top_words

['che double int del una codice questo else bool false return true multiplier api ess
```

```
'div rem camping display href img src padding hover website backgroundcolor placehol
'hello assist soap jms socket jakarta userid today date protocol jmstemplate worksar
'measure claim cell vector outcome cells ignore scientific frequency patients table
'prompt description system service product users self type private detailed image se
'happiness ophelia gas illegal american fantasy blockchain offgrid martial english a
'rdp administrador nat administrator username channel address password giveaway cont
'torch packetid tensor nodeid pytorch putpacketid batch std output edgecost packetpr
'italy italian guarantee audio financial cryptocurrency bitcoin therapy september au
'song album melody apocalypse littlegpt sound michone fingers sunset distant birthda
'clementine luis max john play says nods smile ill fire role looks mimi sumire yeah'
'the and for that with you not this are can from his but its your']
```

```python
from flask import Flask, jsonify, render_template, request
from pyspark import SparkContext, StorageLevel
from pyspark.sql import SparkSession
import pyspark.sql.functions as F
from pyspark.ml import Pipeline
from pyspark.sql.types import ArrayType, StringType
from pyspark.ml.feature import Tokenizer, CountVectorizer, IDF, StopWordsRemover
import re
from transformers import pipeline
from apscheduler.schedulers.background import BackgroundScheduler
```

```python
# Spark session
def defineSparkSession():
    return SparkSession.builder.appName("WILDCHAT-Preprocessing") \
        .config('spark.driver.memory', '32g') \
        .config('spark.executor.memory', '16g') \
        .config('spark.sql.debug.maxToStringFields', 1000) \
        .config("spark.default.parallelism", "24") \
        .config("spark.driver.maxResultSize", "10g") \
        .master('local[8]') \
        .config("spark.serializer", "org.apache.spark.serializer.KryoSerializer") \
        .getOrCreate()
```

```
# Define Flask app
app = Flask(__name__)
spark = defineSparkSession()
summarizer = pipeline("summarization", model="facebook/bart-large-cnn", device = 'mps
result_data = None
keyword = ""
state = ""
country = ""
model = ""

# Custom stop words
custom_stop_words = StopWordsRemover.loadDefaultStopWords("english") + ["\n", "\t", "
global_pipeline = Pipeline(stages=[
    Tokenizer(inputCol='clean_interaction', outputCol='tokenized_clean'),
    StopWordsRemover(inputCol='tokenized_clean', outputCol='swr_clean_tokens', stopWo
    CountVectorizer(inputCol='swr_clean_tokens', outputCol='raw_features', vocabSize=
    IDF(inputCol='raw_features', outputCol='tfidf_features')
])

# Clean text function
@F.udf(StringType())
def cleanText(prompt):
    if prompt:
        clean_text = re.sub(r'[^a-zA-Z0-9\.\*/=:,.&|^%@!# ]', '', prompt)
        clean_text = re.sub(r'([\-*/=:,.&|^%@!#])\1+', r' \1 ', clean_text)
        clean_text = re.sub(r'(\d)([a-z])', r'\1 \2', clean_text)
        clean_text = re.sub(r'([.?])', r' \1 ', clean_text)
        clean_text = re.sub(r'\s+', ' ', clean_text).strip()
        clean_text = clean_text.replace('/', '')
        return clean_text.lower()
    return ''



# Load data
def load_data(spark):
    return spark.read.parquet('/Users/sharan/Desktop/IDMP Data/*.parquet')

# Preprocess data
def prepare_data(df):
    df = df.filter((F.col('language') == "English") &
                   (F.col('toxic') == False) &
                   (F.col('redacted') == False)) \
           .drop('openai_moderation', 'detoxify_moderation', 'hashed_ip', 'header', '
           .withColumn('conversation_explode', F.explode(F.col("conversation"))) \
           .withColumn('prompt', F.col('conversation_explode.content')) \
           .withColumn('state', F.lower(F.col('state'))) \
```

```
            .withColumn('country', F.lower(F.col('country'))) \
            .withColumn('turn_identifier', F.col('conversation_explode.turn_identifier
            .drop('conversation_explode') \
            .fillna({'state': " ", 'country': " "}) \
            .withColumn('clean', cleanText(F.col('prompt'))) \
            .withColumn('clean', F.trim(F.col('clean'))) \
            .drop('conversation')

    groupCols = [col for col in df.columns if col != 'prompt' and col != 'clean']

    return df.groupBy(groupCols).agg(
        F.concat_ws(' --botresp-- ', F.collect_list('prompt')).alias('full_interactio
        F.concat_ws(' ', F.collect_list('clean')).alias('clean_interaction')
    )

# Pipeline definition
def pipeline_definition(df):
    return global_pipeline.fit(df)

# Extract frequent terms
def extract_frequent_terms(tfidf_vector, vocab, threshold=2):
    indices = tfidf_vector.indices
    values = tfidf_vector.values
    terms = [vocab[i] for i, val in zip(indices, values) if val >= threshold and len(
    return terms

# UDF for frequent terms
def frequent_terms_udf(vocab_broadcast):
    @F.udf(ArrayType(StringType()))
    def udf_function(tfidf_vector):
        return extract_frequent_terms(tfidf_vector, vocab_broadcast.value)
    return udf_function

# Main function to process data

def spark_preprocess():
    global result_data  # Declare global variable
    print("Starting Spark preprocessing...")
    main_df = load_data(spark)
    preprocess_df = prepare_data(main_df)
    preprocess_df  = preprocess_df.persist(StorageLevel.DISK_ONLY)
    pipeline_model = pipeline_definition(preprocess_df)
    processed_data = pipeline_model.transform(preprocess_df)


    # Broadcast vocabulary
    vocab_broadcast = spark.sparkContext.broadcast(pipeline_model.stages[2].vocabular
```

```
    # Apply frequent terms UDF
    processed_data = processed_data.withColumn(
        "frequent_terms",
        frequent_terms_udf(vocab_broadcast)(F.col("tfidf_features"))
    ).drop('tfidf_features', 'raw_features', 'swr_clean_tokens', 'tokenized_clean')

    # Cache result data
    result_data = processed_data
    preprocess_df.unpersist(blocking=True)
    result_data = result_data.repartition(8)
    result_data.count()
    result_data = result_data.persist(StorageLevel.DISK_ONLY)
    print("Spark preprocessing completed.")




@app.route("/")
def home():
    return render_template("home.html")


@app.route('/search', methods = ['POST'])
def search_form():
    global keyword, state, country, model
    keyword = request.form['keyword']
    state = request.form['state']
    country = request.form['country']
    model  = request.form['model']

    keyword = keyword.strip().lower() if keyword else ''
    state = state.strip().lower() if state else ''
    country = country.strip().lower() if country else ''
    keyword = keyword.strip().lower() if keyword else ''


    if result_data is None:
        return jsonify({"error": "Data has not been preprocessed yet. Please try agai


    if keyword:
        keyword_result = result_data.filter(F.array_contains(F.col('frequent_terms'),
    else :
        keyword_result = result_data

    if model != 'Both':
        if 'gpt-3.5' in model :
```

```
            keyword_result =  keyword_result.filter(F.col('model').contains('gpt-3.5'
        else:
            keyword_result = keyword_result.filter(F.col('model').contains('gpt-4'))

    if state != '':
        keyword_result = keyword_result.filter(F.col('state') == state.strip().lower(

    if country != '':
        keyword_result = keyword_result.filter(F.col('country') == country.strip().lo

    keyword_result = keyword_result.select(F.col('full_interaction'), F.col('clean_in
    print(keyword, country, state, model)
    keyword_result = keyword_result.withColumn('userprompt', F.split(F.col("full_inte
    result = keyword_result.toPandas().to_dict(orient='records')
    print(result)
    return jsonify(result)


@app.route('/summarize', methods = ['POST'])
def text_summarize():
    text = request.form['text']
    summarized = summarizer(text, min_length = 40, max_length = 1000, do_sample = Fal
    return summarized[0]['summary_text']





# Run Flask app
if __name__ == "__main__":
    spark_preprocess()
    app.run()
```

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Keyword Search</title>
    <style>
        body {
            text-align: center;
            font-family: Arial, sans-serif;
            background-color: #f4f4f9;
            margin: 0;
            padding: 0;
        }
        button {
            background-color: blue;
```

```
    color: white;
    padding: 10px 20px;
    border: none;
    border-radius: 5px;
    cursor: pointer;
}
button:hover {
    background-color: darkblue;
}

#resultsDiv {
    display: flex;
    flex-wrap: wrap; /* Ensures cards wrap to the next row if needed */
    gap: 20px; /* Adds space between cards */
    justify-content: center; /* Centers cards horizontally */
    margin: 20px auto; /* Adds spacing around the results */
    max-width: 80%; /* Limits the total width of the cards container */
}
.card {
    background: white;
    border: 1px solid #ddd;
    border-radius: 5px;
    margin: 10px auto;
    padding: 15px;
    width: calc(50% - 20px);
    box-sizing: border-box;
    box-shadow: 0 2px 5px rgba(0, 0, 0, 0.1);
    cursor: pointer;
    text-align: left;
}
.card:hover {
    background-color: #f1f1f1;
}
.pagination {
    margin: 20px 0;
}
.pagination button {
    background: #007bff;
    color: white;
    border: none;
    padding: 10px 15px;
    margin: 0 5px;
    border-radius: 3px;
    cursor: pointer;
}
.pagination button:hover {
    background: #0056b3;
}
```

```
        .pagination button.disabled {
            background: #ddd;
            cursor: not-allowed;
        }

        .user-prompt {
            color: blue;
            font-weight: bold;
            margin-bottom: 10px; /* Line gap */
        }

        .bot-response {
            color: black;
        }

        @media (max-width: 768px) {
            .card {
            width: 100%; /* On smaller screens, cards take full width */
            }
        }
    </style>
</head>
<body>
<h1>Keyword Search</h1>
<form id="searchForm" action="/search" method="POST">
    <label for="keyword">Keyword:</label>
    <input id="keyword" name="keyword"  placeholder="Enter keyword">
    <br><br>

    <label for="state">State:</label>
    <input id="state" name="state" placeholder="Enter State">
    <br><br>

    <label for="country">Country:</label>
    <input id="country" name="country"  placeholder="Enter Country">
    <br><br>

    <label for="model">Model:</label>
    <select id="model" name="model" placeholder="Enter Model">
        <option value="Both">Both</option>
        <option value="gpt-3.5">gpt-3.5</option>
        <option value="gpt-4">gpt-4</option>
    </select>
    <br><br>

    <button type="submit">Submit</button>
</form>
```

```
<div id="results"></div>
<div class="pagination" id="pagination"></div>

<script>
    const resultsPerPage = 10;
    let currentPage = 1;
    let searchResults = [];

<!--    // Fetch results from the server-->
    document.getElementById('searchForm').addEventListener('submit', async (e) => {
        e.preventDefault();

        const keyword = document.getElementById('keyword').value;
        const state = document.getElementById('state').value;
        const country = document.getElementById('country').value;
        const model = document.getElementById('model').value;

        const response = await fetch('/search', {
            method: 'POST',
            headers: {
                'Content-Type': 'application/x-www-form-urlencoded',
            },
            body: `keyword=${encodeURIComponent(keyword)}&state=${encodeURIComponent(
        });

        if (response.ok) {
            searchResults = await response.json();
            currentPage = 1;
            displayResults();
        } else {
            alert('Error fetching results. Please try again.');
        }
    });

    // Display results with pagination
    function displayResults() {
        const resultsDiv = document.getElementById('results');
        const paginationDiv = document.getElementById('pagination');
        resultsDiv.innerHTML = '';
        paginationDiv.innerHTML = '';

        if (searchResults.length === 0) {
            // If no results, show "No Results" and hide pagination
            const noResultsMessage = document.createElement('p');
            noResultsMessage.innerText = 'No Results Found';
            resultsDiv.appendChild(noResultsMessage);
            return; // Exit the function to prevent rendering pagination
        }
```

```
const start = (currentPage - 1) * resultsPerPage;
const end = start + resultsPerPage;
const currentResults = searchResults.slice(start, end);

currentResults.forEach((result) => {
    const card = document.createElement('div');
    card.className = 'card';
    // Create elements with class names
    const userPrompt = document.createElement('div');
    userPrompt.className = 'user-prompt';
    userPrompt.innerText = result.userprompt;

    const botResponse = document.createElement('div');
    botResponse.className = 'bot-response';
    botResponse.innerText = result.botresp;

    // Append the elements to the card
    card.appendChild(userPrompt);
    card.appendChild(botResponse);

    card.addEventListener('click', async (e) => {
    e.preventDefault();

    const summaryResponse = await fetch('/summarize', {
    method: 'POST',
    headers: {
        'Content-Type': 'application/x-www-form-urlencoded',
    },
    body: `text=${encodeURIComponent(result.clean_interaction)}`,
    });

    if (summaryResponse.ok) {
         const summary = await summaryResponse.text();
         alert(`Summarized Interaction: ${summary}`)


    } else {
        console.error("Error in summarizing ")
        alert('Error fetching results. Please try again.');
    }

    });
    resultsDiv.appendChild(card);
});

const totalPages = Math.ceil(searchResults.length / resultsPerPage);
```

```
        if (totalPages > 1) {
            // Previous button
            const prevButton = document.createElement('button');
            prevButton.innerText = 'Previous';
            prevButton.disabled = currentPage === 1;
            prevButton.className = currentPage === 1 ? 'disabled' : '';
            prevButton.addEventListener('click', () => {
                currentPage--;
                displayResults();
            });
            paginationDiv.appendChild(prevButton);

            // Next button
            const nextButton = document.createElement('button');
            nextButton.innerText = 'Next';
            nextButton.disabled = currentPage === totalPages;
            nextButton.className = currentPage === totalPages ? 'disabled' : '';
            nextButton.addEventListener('click', () => {
                currentPage++;
                displayResults();
            });
            paginationDiv.appendChild(nextButton);
        }
    }
</script>
</body>
</html>
```

B Appendix B: Additional Figures

Include any additional figures or tables that support the analysis.