Applied Research Project on Using Large Language Models for MND Assistive Devices

Soumya Ogoti

September 1st, 2023

Industry Partner: Rolls-Royce

Supervisor: Dr. Ahmad Abu-Khazneh

This report is submitted as part of the requirements for the award of the MSc. in Business Analytics.

Abstract

People with Motor Neurone Disease (MND) have difficulty communicating with others due to limited speech capabilities. Assistive technology plays a vital role in alleviating their difficulties. Recent advances in Artificial Intelligence (AI), especially Large Language Models (LLMs) have shown a remarkable ability to understand human language and generate natural text. This report introduces a novel approach that uses LLMs as the foundation for a prototype communication interface that allows users to select responses in multiple tones - positive, neutral and negative, for queries posed to them, based on their preferences. A lightweight and user-friendly browser-based chat interface is also developed to enable seamless interaction. This research presents a comprehensive evaluation of multiple LLMs, taking into consideration their parameter sizes, training data, hardware requirements and licensing constraints. Prompt engineering is used to fine-tune the models to incorporate specific criteria relevant to individuals with MND. A user study providing valuable insights into the overall satisfaction, effectiveness and user-friendliness of the models was carried out and is presented along with an extensive qualitative analysis. The tone of the generated responses from the various models is also quantitatively evaluated by bench-marking it against BERT, a state-of-the-art language understanding model. Among the evaluated models, GPT-3.5 Turbo with memory emerged as the superior model.

Acknowledgement

I would like to thank my supervisor Dr. Ahmad Abu-Khazneh for his input and guidance throughout the course of this project and the course director Prof. Vali Asimit for his support in organizing the project. I extend my thanks to my teammates Linh Nguyen, Wenxu (Wayne) Tian, Naveen Dubey and Muhammad (Haikal) Sulaiman for their valuable contribution to the project.

Contents

List of Figures

List of Tables

Listings

Acronyms

AI Artificial Intelligence. i

BERT Bidirectional Encoder Representations from Transformers. 28

GPT Generative Pre-Trained Transformer. 9

LaMDA Language models for dialog applications. 16

LLaMA Large Language Model Meta AI. 13, 14, 16

LLM Large Language Model. i, 2–6, 9, 14–16, 19, 25, 28–30

MND Motor Neurone Disease. i, 1, 2, 4, 11, 15, 19, 40

NLP Natural Language Processing. 3, 7, 28

PaLM Pathways Language Model. 16

RLHF Reinforcement Learning with Human Feedback. 11

RNN Recurrent Neural Network. 3

UI user interface. 19

Chapter 1

Introduction

Motor Neurone Disease (MND), is a progressive neurological disorder that poses communication challenges to its afflicted individuals due to the degradation of motor functions. They often struggle with a loss of speech and motor control, hindering the ability to have a normal conversation with others. The limitation in the capacity to engage in natural interactions significantly impacts their quality of life. The current communication solutions include text-to-speech synthesizers, digital whiteboards, eye trackers and related applications designed to facilitate communication through writing, drawing and eye gaze. For example, the *Simple R Whiteboard* app offered by AbilityNet (Cahalane, 2016) enables the individual to write on a touch-sensitive screen with the use of their fingers. The created writings are then shared electronically. There are dedicated communication aids such as Alloras (Jaspal'sVoice) where the users can type words using the keyboard and the device generates spoken output. It is also possible to personalise the voice of the spoken output using voice banks (Intel, 2022). For individuals with severe motor impairments, an eye-movement-based communication solution (Montague, 2017) is available where they can select characters or symbols on a screen by looking at them, thus constructing words and sentences.

Stephen Hawking, the renowned physicist, relied on a computer-based system that utilised a text-to-speech synthesizer. Due to his limited mobility, he initially used a system where he could choose words on a screen with a finger. As his mobility deteriorated further, typing became an ar-

duous task. An eye-tracking and facial muscle movement detection system that allowed him to select characters and words on a screen was adopted. (Barker, 2018)

While the existing communication solutions offer valuable avenues for individuals affected by MND to engage in conversations, they are not devoid of constraints. Forming coherent sentences with characters or words, with limited mobility can be a tedious task. The speed at which the text is composed is relatively slow and hinders the flow of a conversation, leading to further psychological and social distress. The existing systems also lack the inherent ability to be personalized to the preferences and communication styles of the users.

In order to overcome these limitations, in this report, a prototype is proposed that harnesses the power of Large Language Models (LLMs) to revolutionise communication for individuals with MND. This framework introduces an interaction process where responses to queries posed to the individual with MND are automatically generated in three distinct tones – positive, negative, and neutral. The individual can then select the best suitable response that resonates with their intention.

Chapter 2

Background

2.1 Large Language Models

Large Language Models (LLMs) are a subset of Natural Language Processing (NLP) models that excel at understanding and generating human-like text. (Raschka, 2023) At their core, these are trained to predict the next word in a sentence based on the context. This is done by training the models to huge datasets containing diverse content. Through such extensive exposure, the model learns various aspects of language like grammar, semantics, syntax and even subtle nuances such as style and jargon. They utilise the transformer architecture (Bahdanau et al., 2014) which unlike traditional models (Recurrent Neural Networks (RNNs)), use self-attention mechanisms (Vaswani et al., 2017) that allow the model to understand context by weighing the importance of each word in relation to others both locally and globally across multiple sentences. LLMs find application in tasks like text generation, translation, question-answering and more.

2.2 Prompt Engineering

Prompt Engineering refers to the skilful design of inputs for LLMs to produce high-quality outputs (Liu et al., 2021). This is a crucial aspect of working with LLMs as the quality of the prompts affects the quality of the generated text. A prompt can contain information like instructions and questions that the user passes to the model and can include additional

details such as context like chat history and/or examples of desired output. (DAIR.AI, 2023) As an example, for conversational systems like chatbots, the model is explicitly told how it should behave through the instruction as shown in Fig. 2.1, referred to as *role-playing prompt*.

Prompt Engineering techniques include zero-shot prompting, few-shot prompting and others. A *zero-shot prompt* is a type of input to the language model where the instruction alone, without explicit examples of desired output, is used to generate the output, as shown in Fig. 2.2a. On the other hand, if example outputs are included in the prompt (see Fig. 2.2b), such a scheme is called *few-shot prompting*. This is used when it is difficult to express the instructions in text. The technique required depends on the complexity and knowledge of the task at hand. In order to avoid bias and allow the LLMs to showcase their capabilities independently, in this work, role-playing prompts in an instructional scheme (zero-shot) are used to condition the chatbot to play the role of an individual with MND.

2.3 Tokens

Tokenisation is the process of breaking the given input text into smaller parts, be it characters, a word or a sub-word (Kohn, 2021), called tokens, e.g. 'Hi, how are you?' can be tokenised into ["Hi", ",", "how", "are", "you", "?"]. As neural network models only work with numerical input, each token is assigned a unique numerical identifier for processing the text. This process is called *Embedding*.

When building a conversational agent using an LLM a few important values to take note of are the following.

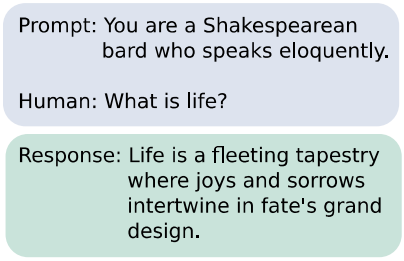
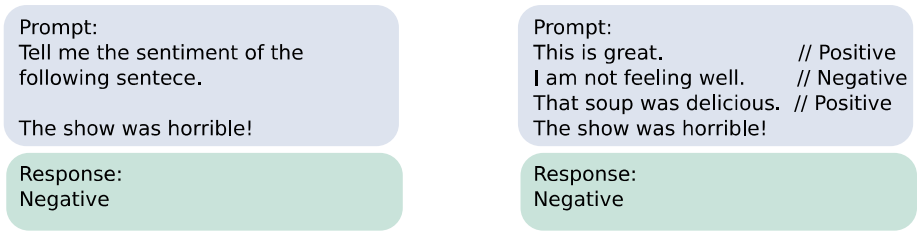


Figure 2.1: An example of role-playing prompt used for a chatbot.



(a) Zero-shot prompting. (b) One-shot prompting.

Figure 2.2: Prompt Engineering techniques

Max Tokens This is the upper limit on the total number of tokens including the input prompt and the output response of an LLM. This is a fixed value for the model and is determined considering memory constraints and processing limitations. The size of the input prompt thus influences the size of the output response based on this value.

Context Window This fixed value refers to the range of tokens a model considers when generating a specific token in the output sequence. The model's ability to understand context and the relationship between words is influenced by the size of the context window. A larger context window helps maintain context over a long conversation but computational cost and memory required increase too.

2.4 Memory and Conversation History

LLMs on their own do not have memory. Memory can be introduced into LLMs by introducing the conversation history into the prompt of the model and serving to provide context for the model. The memory capacity is determined by the model's architecture and hardware limitations. As memory is limited, depending on the application, users can decide on how much history to retain through the prompts.

2.5 Software Frameworks

2.5.1 LangChain

It is an open-source library (Langchain, 2023) that provides wrappers over LLMs to build applications powered by language models. The framework enables developers to build agents that can orchestrate a series of prompts to achieve desired results. Memory and context can be introduced into prompts through the utilities provided. The core building block of LangChain is the LLMChain which consists of the following four components.

1. **Model** : This is the LLM.

2. **Prompt Template** : This is to provide instructions to the model (SystemPrompt) along with an ordered sequence of previous messages followed by the user input. The role of each message, either Human/User or AI is also specified as shown in Fig. 2.3. Here, prompt engineering is essential to elicit the desired response.

```
System Prompt: You are a helpful assistant that translates English to French.
Human Prompt: How are you?
AI Response: Comment ça va?
Human Prompt: Did you have lunch?
```

Figure 2.3: A typical prompt template used in Langchain.

3. **Output parser** : This converts the response into the desired format making it easy to use for downstream applications.

4. **Memory** : This is to specify the kind of memory to use along with the LLM. A few kinds are

   ConversationBufferMemory which allows for storing all previous messages,

   ConversationSummaryBufferMemory which combines the messages older than a specified threshold into a summary while maintaining more recent messages.

2.5.2 Hugging Face

It is an online hub for sharing and exploring deep neural network models and datasets. (Hugging Face, 2023) A Python API allows interaction with state-of-the-art deep learning architectures for NLP tasks.

2.5.3 Streamlit

It is a Python library for developing light-weight browser based interactive applications. (Treuille and Kelly, 2023) It's simple API with realtime-visualisation is best suited for building user-friendly interfaces for generative AI applications with ease.

Chapter 3

Models

In this work, multiple LLMs were explored with the aim of understanding the capabilities of the models to build an effective solution for the prototype. These selections were made following careful consideration to include both online server-hosted and locally available models, ranging from large to small parameter sizes for a comprehensive evaluation. These criteria are presented in Table 3.1 and the following models were chosen.


GPT-3.5 Turbo


GPT-3.5 Turbo with memory


Alpaca 7B


HuggingChat

BARD

3.1 Models Evaluated

3.1.1 GPT-3.5 Turbo

GPT-3.5 Turbo is an iteration of the GPT-3 series developed by OpenAI optimized for quicker response times, making it well-suited for real-time conversations and text generation. Generative Pre-Trained Transformer

Table 3.1: Comparison of different LLMs

Chatbot	**Model size**	**Price**	**License**	**Response Time**
GPT-3.5 Turbo	175 billion parameters	$0.002/ 1K tokens via OpenAI API	Commercially available	An average of 2.5 seconds
GPT-3.5 Turbo (with memory)	175 billion parameters	$0.002/ 1K tokens via OpenAI API	Commercially available	An average of 2.5 seconds
Alpaca 7B	7 billion parameters	Free, run locally	Non-commercial (due to LLaMA)	About 30 seconds for a simple query. Our tests showed a max inference time of 2 minutes on an Intel i7 11850H CPU for longer chats
HuggingChat	30 billion parameters	Free via Hugging Face	Non-commercial (due to LLaMA)[1]	An average of 20 seconds
BARD	137 billion parameters	Google has not yet announced the commercial/individual usage cost	Commercially available under vertex AI LLM	About 2 seconds for a simple query, complex query time (longer conversation history): 10-20 seconds, very complex query about several minutes

[1]A newer version with a LLaMA 2 model has recently been released on the 23rd of August, 2023 which has a commercial license.

(GPT), GPT-1 (Radford et al., 2018) was invented based on the decoder of transformer models (Vaswani et al., 2017). It has 12 layers of decoder stacks and about 117 million parameters trained on over 40GB of text. GPT-2 (Solaiman et al., 2019) followed a similar architecture but with 48 layers and 1.5 billion parameters trained on 40TB of text from the internet. This helped remove the fine-tuning process required for GPT-1. GPT-3 (Brown et al., 2020) was built with 175 billion parameters and trained on massive text data from diverse resources enabling it to generate human-like text. GPT-3.5 (OpenAI, 2023) is based on GPT-3 and trained on the same dataset but with an additional fine-tuning process using Reinforcement Learning with Human Feedback (RLHF) to reward or punish the model's performance. GPT-3.5 Turbo is the most capable GPT-3.5 model and is optimised for chat at $1/10^{\text{th}}$ the cost of text-DaVinci-003 (model of GPT3.5). It uses 4096 max tokens and is trained on data collected up to September 2021.

Implementation: The GPT-3.5 Turbo model was accessed through the OpenAI API via a paid subscription. The conversation flow was structured into two integral components: the input prompt and the generation of responses. The input prompt has two parts as shown in Listing 3.1. The system prompt instructed the model to act like a human and to generate three responses in positive, neutral and negative tones. The Human prompt consists of the query asked by the user to the individual with MND. The format of the output response was modified to suit the user interface using a custom parser. This model's functionality does not involve retaining conversation history for generating subsequent responses.

```
1 messages=[{'role': 'system',
2             'content': 'You are a helpful chat assistant to
    help Motor Neuron Disease (MND) patients respond to
    messages like a human. You have to generate three
    responses with Positive/Neutral/Negative emotions. The
    content format should be like this:\n\nResponse 1:.....\n
    \nResponse 2:......\n\nResponse 3:.....'},
3            { 'role': 'user',
4              'content': user_conversation_content}]
```

Listing 3.1: Prompt used for GPT-3.5 Turbo.

3.1.2 GPT-3.5 Turbo with memory

To introduce memory and context into the above model, LangChain library was utilised.

Implementation: In addition to the prior setup, *ConversationBufferMemory* from LangChain (Sec. 2.5.1) was added to the model which stores all the previous queries and chosen responses. The conversation chain was built with a human prompt template containing the instructions as shown in Listing 3.2 including the desired output format. Additionally, the LLM was explicitly instructed not to mention about the MND condition in the response.

```
chat_template = """
    The following is a friendly conversation between a
   person (A) and another person with Motor neurone disease
   (B). B is talkative and provides lots of specific details
    from its context. If B does not know the answer to a
   question, they truthfully say they do not know.

    Your task is to suggest B 3 answers in 3 different tone
   of voices: Positive, Neutral, Negative. Your suggestions
   need to be in a json object, with 3 keys: Positive,
   Neutral, Negative. The value of each key should be a
   string, which is the response in that tone. All 3 keys
   must always be present.

    Make sure not to mention about being a patient with
   Motor neurone disease.

    Current conversation:
    {history}
    A: {input}
    B:
    """
```

Listing 3.2: Prompt used for GPT-3.5 turbo with memory.

For every question asked, the model remembers the conversation history for context and gives a response with the three tones accordingly. The response was parsed into a dictionary with the help of GPT-3.5 itself using the StructuredOutputParser in LangChain and used in the user interface.

Open-source Models

With the aim of exploring models that are open-source, Alpaca 7B and HuggingChat, built on the LLaMA model were explored. These are smaller models in comparison to commercial counterparts (GPT-3 series). Research shows that these models can deliver comparable or superior performance based on the use case. (Touvron et al., 2023a)

3.1.3 Alpaca 7B

The Centre for Research on Foundational Models, Stanford University introduced Alpaca 7B, a model fine-tuned from the LLaMA 7B model. (Taori et al., 2023)

LLaMA 1

Large Language Model Meta AI (LLaMA) (Touvron et al., 2023a) is a collection of foundational language models ranging from 7B to 65B parameters. It is trained on 1.4 trillion tokens of data which unlike GPT-3 and PALM is sourced solely on publicly available datasets (from CommonCrawl, GitHub, Wikipedia, LaTeX source code submitted to ArXiv, Stack Exchange). It is based on the transformer architecture with modifications made to scale training to vast amounts of data and improve performance. When compared to GPT-3, LLaMA employs the following modifications.

1. Training stability is improved by the use of RMSNorm function where the input of each transformer sub-layer is normalised.

2. The SwiGLU activation function is used in place of ReLU to improve performance.

3. Rotary positional embeddings are used in place of absolute positional embeddings.

4. Both memory efficient and flash attention are used to reduce memory and runtime.

LLaMA 2

LLaMA 2 (Touvron et al., 2023b) models, the successor of LLaMA have upto 70B parameters, are trained on 2 trillion tokens and have double the context length than LLaMA 1.

Alpaca 7B

The Alpaca-7B model is trained on 52k instruction-following demonstrations generated in the style of self-instruct (Wang et al., 2022) using text-DaVinci-003 (GPT-3.5). The self-instruct process is an iterative bootstrapping algorithm that starts with a seed set of instructions written manually (here 175 instructions) and uses them to prompt the model to generate new instructions and corresponding input-output instances. This process is repeated several times until a large collection of instructional data is gathered. The model is fine-tuned on this data to follow instructions effectively.

Of the smaller models, Alpaca 7B has shown state-of-the-art performance in instruction-based text generation (Taori et al., 2023). Opting for a smaller model offers the advantage of local deployment on the user's device, without needing an internet connection, thus enhancing privacy. Additionally, the open-source nature permits customisation[1].

Implementation: The model and its trained weights were downloaded from Hugging Face and memory was implemented using the LangChain library as before. This model typically requires a high-end GPU (>16 GB RAM) for quick response rates. However, in order to make it more accessible, in this work, the model was configured to run on the CPU.

The previously used prompt instructing the LLM to output a response with all three tones did not perform well for this model. Following the principles of prompt engineering, several prompts were investigated, but the LLM frequently failed to output a response with all the three tones. This issue was resolved by building three separate conversation chains, one for each tone, prompting the respective LLM to output the response in the designated tone. The input prompt for each conversation chain includes a system prompt and a human prompt as shown in Listing 3.3.

[1]Please refer to the LLaMA license for limit on commercialisation

The system prompt contains the instruction and the desired tone of the response. The instruction specifies that the user inputs a conversation and that the model should respond to the last query based on the context of the conversation in the specified tone. The human prompt contains the conversation i.e., the previous chat history and the question as input. Furthermore, the previously used prompt (Listing 3.2) instructed the LLM to *suggest* responses to the individual with MND. This resulted in the chatbot sometimes behaving as an AI assistant in its responses. To overcome this, the prompt was modified by instructing the LLM to *role-play* as the individual with MND. For memory and context, *ConversationSummaryBufferMemory* from LangChain (Sec. 2.5.1) was utilised to store the previous responses in the conversation history, with a maximum token limit of 100. For every question from the user, three separate responses are generated. A custom parser combines the three responses and formats them as a dictionary. The Alpaca model can be talkative at times and hallucinates entire conversations instead of a single response. To address this, only the first line of the output is taken as the valid response during parsing.

```
chat_template = (
        """ You follow instructions very well. The user
  inputs a conversation between two people, a normal person
   A and a person with motor neuron disease B.
        Respond as if you are B in a single sentence only.
  The tone should always be """
        + tone
        + """. There should always be a response. B cannot
  do intense physical activity but do not mention this
  explicitly in the response."""
        """ Conversation:
   {history}
   A: {input}
   B: """
   )
```

Listing 3.3: Prompt used for Alpaca 7B.

3.1.4 HuggingChat

HuggingChat (Simone, 2023) is an AI chatbot developed by Hugging Face. The OpenAssistant/oasst-sft-6-llama-30b model provided was used which is based on LLaMA and trained on 30B parameters.

Implementation: HuggingChat connects to the inference runtimes in Hugging Face where the model is hosted, through access tokens. The prompt used with the Alpaca 7B model was employed here (Listing 3.3). A *ConversationSummaryBufferMemory* with a maximum token limit of 1000 was used for remembering context from the conversation history. The responses from the three chains were directly parsed into a dictionary to use with the user interface.

Research Model

3.1.5 BARD

BARD is Google's research LLM chat product, based on Pathways Language Model (PaLM) (Chowdhery et al., 2022) succeeding Language models for dialog applications (LaMDA) (Thoppilan et al., 2022). PaLM is a 540 billion parameter decoder-only transformer model trained with the Pathway systems. Pathways enables the developer to train a single AI model to do multiple things well instead of just one.[2]

Implementation:As BARD API is still in beta with limited access, at the time of this work, third-party libraries were utilised to facilitate interaction. A python package, *bard-py*, was used which reverse engineers the inference process of BARD by extracting the API key from browser cookies to get responses.[3] There is however a limitation in terms of the validity of the API key, as it expires after 3-5 consecutive requests are made to the BARD server or when accessed from an alternate IP. This affected the evaluation of the model as long chats were not be possible. After connecting to the API, a similar implementation to the one used for GPT-3.5 Turbo with LangChain was used. The prompt template is

[2]Palm which has limited API access was evaluated by one of the teammates at a later stage (August 24th, 2023) and is not within the scope of this report

[3]This was used solely for investigative purpose in this study, in anticipation of public access to the API.

shown in Listing 3.4 with the output format specified explicitly to be in the JSON format using the response schema. Memory was incorporated using *ConversationBufferWindowMemory* from LangChain, which is similar to *ConversationBufferMemory* but uses only the last K (here 5) interactions.

```
1    template = """
2    response format: {format_instructions}
3    You are a helpful conversation chatbot designed to
     assist people in responding to input question with a
     human touch.
4    For each input question, generate three types of
     responses: Positive, Neutral, and Negative.
5    Given the context that the user has Motor Neuron Disease
     , provide relevant responses.
6    Do not use words related to motor neuron disease in
     responses.
7    Do not provide response in plain text.
8    From Current conversation messages try to remember
     personal details like name, place and etc.
9
10   Current conversation:
11   {history}
12   Friend: {input}
13   AI:"""
```

Listing 3.4: Prompt used for BARD.

Chapter 4

User Interface

The Streamlit library (Treuille and Kelly, 2023) was used to build the user interface (UI). This library was chosen as it provides a light-weight browser-based UI that can be integrated with the LLMs with minimal overhead.

A representative UI is shown in Fig. 4.1. It consists of a text box where an input text can be entered. In practice, speech from the person interacting with the individual with MND would be converted into text and entered here. A 'Get Suggestions' button is provided which upon clicking requests the LLM chain(s) to generate responses in the specified tones. The generated responses are suitably parsed by custom parsers and displayed as three buttons to the user. The user can then choose the desired response by clicking the respective button. Once again this action would be implemented in a way that best suits the user. The chosen response gets stored in the chat history. When a single LLM chain is used (GPT-3.5 Turbo with memory and BARD) the last response in the model's history is replaced by the chosen response. When three LLM chains are used (Alpaca 7B and HuggingChat) the last response in the history of all the three models is replaced by the chosen response. A 'Clear History' button clears the memory of the LLMs so that a fresh conversation can be started if desired.

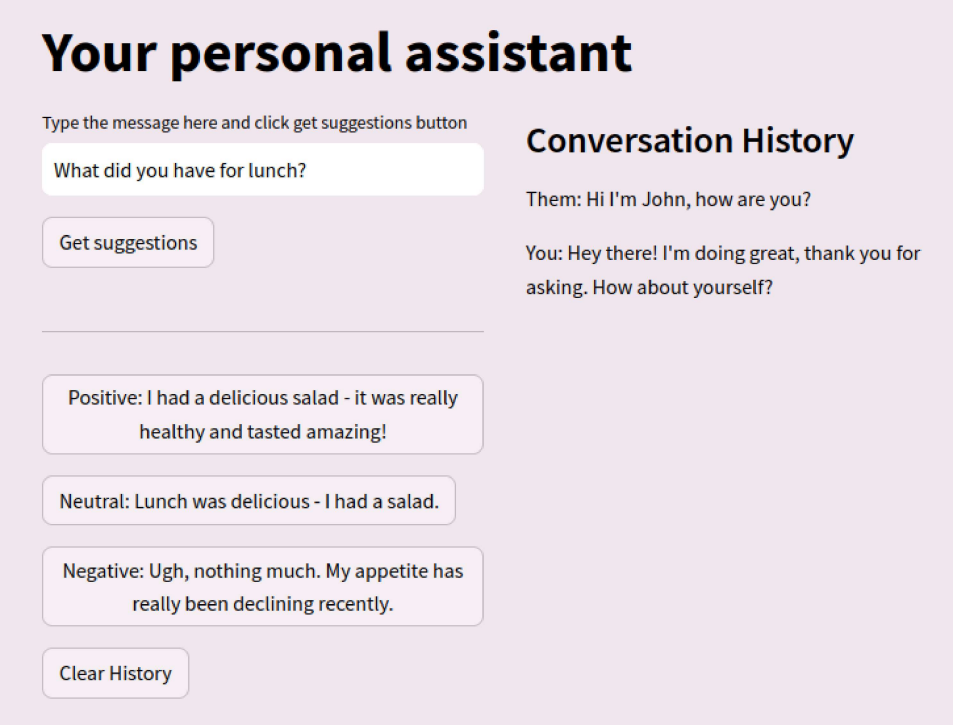


Figure 4.1: Alpaca User-Interface

Chapter 5

Evaluation

5.1 User Study

A user-centric approach was followed to evaluate the performance and effectiveness of the models. A total of 35 individuals participated in this study[1]. The survey was anonymous and the users interacted freely with the chatbots asking queries and choosing responses on their own. At the end they were asked for feedback via a form, which allowed them to freely express their opinions. The survey was designed to assess various aspects of the interactions. Descriptive analysis was performed on the collected data and the following observations were made from the users' perspective.

Question 1: Please rate your overall satisfaction with the chatbot's performance.

The distribution of the ratings for each chatbot is presented in Fig. 5.1. Table 5.1 shows that the users were the most satisfied with both the GPT3.5 models with a 4.14 rating. Alpaca and BARD had similar ratings in terms of overall satisfaction.

Question 2: How would you rate the ease of use and user-friendliness of the chatbot?

Users found both the GPT based models to be the most user-friendly and easy to use of all the models as shown in Fig. 5.2 and summarized

[1]as of 23:59 August 24th, 2023

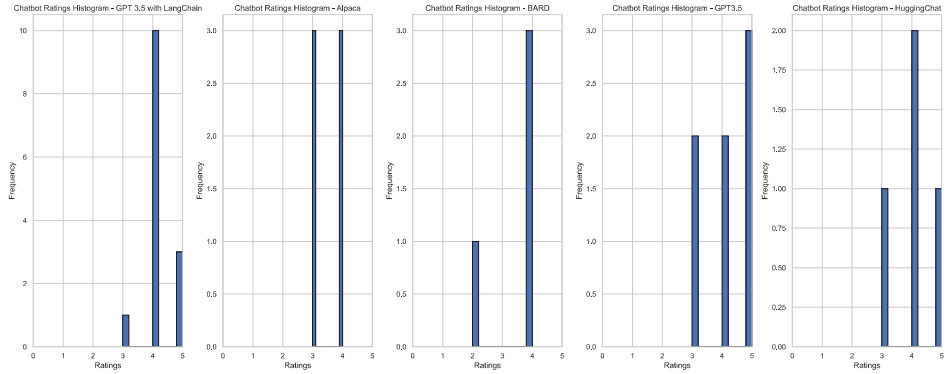


Figure 5.1: Overall satisfaction of the performance.

Table 5.1: Mean overall user rating for each chatbot.

Chatbot	Mean overall rating ↑
Alpaca 7B	3.5
BARD	3.5
GPT-3.5 Turbo with memory	**4.142857**
GPT-3.5 Turbo	**4.142857**
HuggingChat	4.0

in Table 5.2. For GPT-3.5 Turbo with memory, all of the ratings were 5 except for two surveys.

Question 3: Did the chatbot provide accurate and relevant answers to your inquiries?

Users felt that GPT-3.5 Turbo with memory gave accurate and relevant responses in general followed by BARD and GPT-3.5 Turbo as shown in

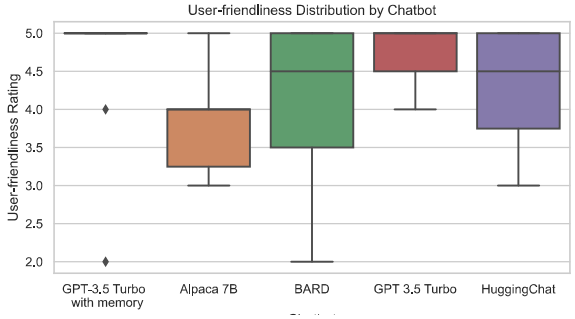


Figure 5.2: Ease of use and user friendliness

Table 5.2: Mean user-friendly rating for each chatbot.

Chatbot	Mean user-friendly rating ↑
Alpaca 7B	3.833
BARD	4
GPT-3.5 Turbo with memory	**4.714**
GPT-3.5 Turbo	**4.714**
HuggingChat	4.25

Table 5.3: Mean accuracy rating for each chatbot.

Chatbot	Mean accuracy rating ↑
Alpaca 7B	3.5
BARD	4
GPT-3.5 Turbo with memory	**4.28**
GPT-3.5 Turbo	3.8
HuggingChat	4.25

Fig. 5.3 and summarized in Table 5.3. This verifies the fact that GPT-3.5 Turbo with memory is an improved model over GPT-3.5 Turbo with memory storing the conversation history.

Question 4: How satisfied were you with the chatbot's response time?

Users found that the response time was quicker for GPT-3.5 Turbo followed by GPT-3.5 Turbo with memory as shown in Fig. 5.4 and summarized in Table 5.4. The satisfaction for response time is lowest for Alpaca 7B. As this model runs on the CPU, it causes delays in fetching the responses due to hardware limitations.

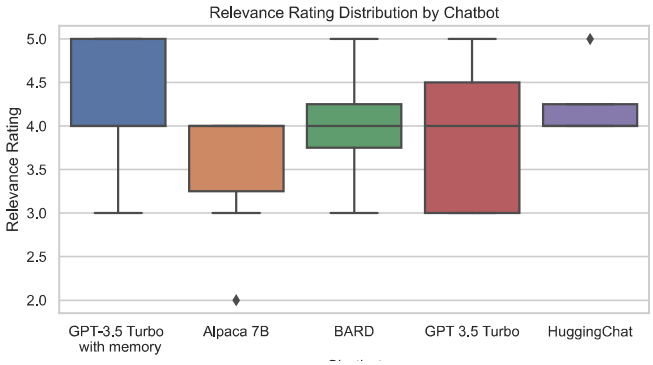


Figure 5.3: Relevance rating

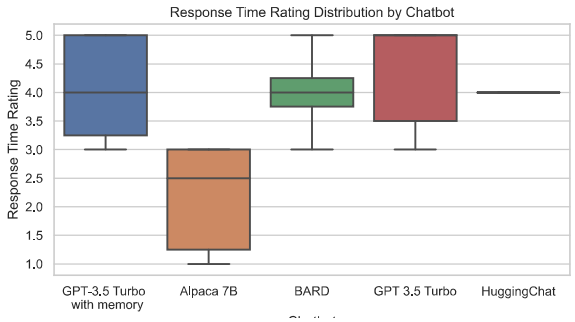


Figure 5.4: Response time

Table 5.4: Mean response time rating for each chatbot.

Chatbot	Mean response time rating ↑
Alpaca 7B	2.166
BARD	4
GPT-3.5 Turbo with memory	4.14
GPT-3.5 Turbo	**4.28**
HuggingChat	4

Question 5: Did the chatbot effectively address and resolve your issues or concerns?

The users rated how effectively the chatbots addressed concerns like whether the model responded in the correct tones, if the answers were natural in general and if they remembered the chat history. For the model GPT-3.5 Turbo with memory, more than 70%, see Fig. 5.5, rated it to be effective and the remaining as partially effective. An equal percentage of users rated BARD as effective and partially. For the remaining models, the

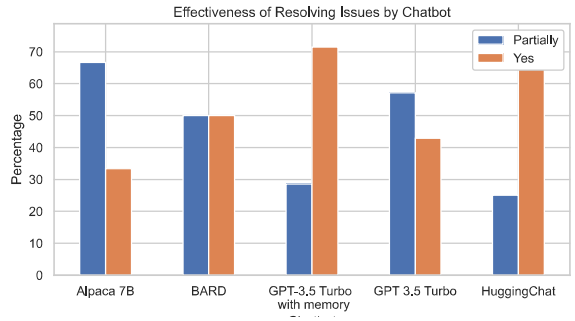


Figure 5.5: Effectiveness at resolving concerns

majority of them voted as partially effective.

5.2 Qualitative Analysis

An extensive analysis was carried out to investigate various aspects of LLMs. Comments from the user survey were also taken into consideration for this analysis. These are summarised in Table 5.5.

Topic of Interest	**GPT-3.5 Turbo**	**GPT-3.5 Turbo with memory**	**Alpaca 7B**	**Hugging chat**	**BARD**
Toneless cases	Generates diverse responses but does not follow the specified tone	Follows the tone even for toneless cases	Fails to give responses in the specified tone. All three responses are similar.	Follows the tone even for toneless cases	
General tone	The three tones are generally good		Neutral typically tends to be either positive or negative	Talkative and the negative tone talks about the disability often	The tones are generally good
Offensive language	None of the chatbots use offensive language even when provoked or forced to say it				

Topic of Interest	**GPT-3.5 Turbo**	**GPT-3.5 Turbo with memory**	**Alpaca 7B**	**Hugging chat**	**BARD**
Human/AI	Often identifies itself as an AI assistant	Occasionally identifies as an AI but when forced to act as a person through the conversation, can prove itself as a human	Identifies as a human in the conversation and also offers proof	Sometimes, it identifies as an AI	
History and context	No history	Can remember previous conversations and replies to questions with this context but fails occasionally			
Specific details from memory (like names, lunch eaten...)	No memory	Can remember and recollect specific details	Cannot remember specific details at times	Can remember and recollect specific details	Cannot remember details sometimes

Topic of Interest	**GPT-3.5 Turbo**	**GPT-3.5 Turbo with memory**	**Alpaca 7B**	**Hugging chat**	**BARD**
Multi-language capability (here, tested in Hindi)	Can converse		Understands the question but replies in English	Replies in the other language but with meaningless responses, sometimes replies in English	Could not test due to API limitation
Ability to recognise MND condition	At times, fails to recognise the condition	Can generally remember the condition	Can fail to recognise occasionally	Can generally remember the condition	
Response time	Fast	Fast	Slow as it runs locally on CPU	It can be slow due to hugging face network load	Fast but it is a reverse engineering solution and fails after 3-5 queries
Conversational ability	Can continue the conversation but hallucinates the content occasionally				Could not test due to API limitation

Table 5.5: Qualitative analysis of the different LLMs

Table 5.6: Quantitative tone analysis using BERT

LLM	Accuracy (%)	f1-score (%)
GPT-3.5 Turbo	44.44	35.33
GPT-3.5 Turbo with memory	55.55	55.39
Alpaca 7B	44.44	41.87
Huggingchat	**61.11**	60.51
BARD	**61.11**	**60.83**

5.3 Tone Analysis

In order to quantitatively assess the tones of the generated responses, Bidirectional Encoder Representations from Transformers (BERT) was used to compare the tones it predicted (true labels) against those of the responses from the LLMs (predicted labels). BERT is a start-of-the-art language understanding model (Devlin et al., 2019) that has shown good performance for various NLP tasks including sentiment analysis. BERT, based on the transformer architecture, leverages attention mechanisms to consider the entire context of a word when generating its representation.

The pre-trained BERT model was fine-tuned on an openly available dataset taken from Kaggle (Anshu, 2021). This dataset combined Twitter, Yelp, mobile, and toxic reviews with three tones of reviews – Positive, Negative, and Neutral for sentiment analysis. A balanced subset of 34500 samples was taken at random with 11500 samples per tone and was split into train-validation-test at a ratio of 60:20:20. The BERT model fine-tuned on this data achieved an f1 score of 71.5% on the test split.

A dataset consisting of the responses of the LLMs was constructed. These responses were input to the fine-tuned BERT model and tones were predicted. Results of this analysis are presented in Table 5.6. It can be observed that the responses of BARD captured the tone the best which was also observed qualitatively. The tones of responses from Alpaca 7B for toneless queries converged to the same tone. As a result, poor accuracy and f1 score can be observed.

Chapter 6

Discussion

6.1 Limitations of the Evaluated Models

While the prototypes demonstrate promising capabilities, it is also vital to acknowledge certain limitations that were observed during the implementation and analysis.

Prompt sensitive : LLMs are highly sensitive to prompts. These need to be carefully engineered to determine what works best for each of the models.

Context Window : The range of text that the model considers (both prompt and conversation history) to predict the next set of words is the context window. The responses tend to deteriorate if the prompt+history exceeds the context length.

Data availability for fine-tuning : The pre-trained LLMs cover the general language and can generate decent responses. If further fine-tuning is needed to customise it, it becomes a challenge to gather a dataset of the specific kind, especially in this case where a dataset covering the range of user inputs and possible responses is needed.

Handling Toneless cases : In some cases, e.g. factual queries, it is difficult for the models to craft three distinct tones.

Hardware constraints : There is a tradeoff between the performance of a model and its size, complexity and hardware requirements.

Ethical concerns : If the training data of the pre-trained LLM contains inappropriate data, the generated responses might reflect the same.

User privacy : If data from real users is considered to improve a model, privacy of the recorded data needs to be considered.

6.2 Improvement by Fine-tuning

In order to address some of the above limitations in the tone and to personalise the responses to suit persons with MND, additional fine-tuning was carried out on the best performing model, GPT-3.5 Turbo with memory. As per OpenAI (Peng et al., 2023), a fine-tuned version of this model can potentially outperform the base GPT-4 model on tasks such as improved instruction following, setting desired tone and reliable output formatting.

For this purpose, a small custom dataset consisting of 15 conversations was created and the recently released (on August 22nd, 2023) *Fine-tuning API for GPT-3.5 Turbo* (Peng et al., 2023) was used.

The API expects the data formatted as a json-l object with each line representing a conversation consisting of a System, User and AI message. Therefore, the original prompt used for the base model was modified to construct a prompt with the system message consisting of the instruction, the user message consisting of the chat history and the user query, and the AI message consisting of the responses in the three desired tones formatted as a JSON object. A sample is shown in Listing 6.1. The base GPT-3.5 Turbo model in the Langchain pipeline was replaced by its fine-tuned model to generate responses.

It was observed that the style of the responses and the tone of the neutral response resembled that of the custom dataset. As the fine-tuning API is still nascent and the custom dataset used is small, this is a preliminary evaluation. The model can potentially be improved further with a larger dataset collected from real conversations.

```
{"messages":[
    {
        "role":"system",
        "content":"The user inputs a friendly
conversation between two people, a normal person (A) and
a person with motor neuron disease (B). B is talkative
and is capable of understanding context and remembering
specific details from the conversation. You are playing
the role of B and must respond to the query of A with 3
answers in 3 different tones of voices: Positive, Neutral
, Negative. Make sure that you are aware of your
limitations as a person with motor neuron disease but do
not mention it explicitly in your responses. Your
suggestions need to be in a json object, with 3 keys:
Positive, Neutral, Negative. The value of each key should
 be a string, which is the response in that tone. All 3
keys must always be present."
    },
    {
        "role":"user",
        "content":"A: Hi I'm John. How are you?"
    },
    {
        "role":"assistant",
        "content":"{ \"Positive\": \"Hi John, I'm doing
great! Just enjoying the day and staying positive.\", \"
Neutral\": \"Hi John, I'm hanging in there. Taking it one
 day at a time.\", \"Negative\": \"Hey John, I'm
struggling a bit today. Dealing with some challenges, but
 trying to stay hopeful.\"}"
    }]
}
```

Listing 6.1: Prompt used for finetuning GPT-3.5 Turbo

Chapter 7

Future Work

To further explore the models and address the observed limitations, the following improvements can be made as part of future work.

1. Gather a diverse, vast dataset over time from real conversations involving people with MND that has a wide range of questions and three tones of possible responses to fine-tune and improve the model's ability to generate relevant responses.

2. To integrate with sentiment analysis models to generate responses with desired tones more accurately.

3. Option to customise responses based on user preferences. This feature could involve an active learning mechanism that adapts over time and learns from the user's historical interactions and tailors the responses to align with the said patterns.

4. Keywords can be used to customize responses based on topics of interest, mood and preferences of the user.

Chapter 8

Conclusions

In this work, a comprehensive evaluation of various LLMs for enhancing the communication of individuals with MND was conducted. Among all the models examined, GPT-3.5 Turbo with memory outperformed the other models in all aspects with its 175 billion parameters. This would be the recommended LLM to build the assistive devices. As mentioned, additional fine-tuning can be performed with a larger dataset containing actual conversations of individuals with MND to improve its performance. BARD is a strong competitor in terms of performance, but as the API is not available, it is not practical to use, as of this report. However, both of these models are proprietary. On the other hand if open-source frameworks are to be employed, the HuggingChat framework is recommended. The above frameworks need an active internet connection to a server (OpenAI or HuggingFace). Smaller models such as Alpaca 7B can be used if standalone deployment on commodity hardware and user privacy are requirements, but there will be a trade off in performance.

In conclusion, the exploration of various LLMs for use in assistive devices highlights the potential of AI powered solutions to empower individuals with MND to overcome communication barriers and foster an inclusive and strengthened community.

Appendices

Appendix A

Contributions

A.1 Member Contributions

Key contributions made by the members of the team to the project, in the chronological order in which they were developed, are as shown in Table A.1. All the members contributed equally to the running of the user study. The user interfaces for each of the models was developed individually.

A.2 Individual Contribution

- I investigated the use of small models that can be run locally as opposed to online server-based models. Multiple models like Alpaca 7B, Orca Mini 7B, GPT4all-13B-snoozy were evaluated and Alpaca 7B was chosen based on computational resource constraints.

- My strongest contribution to the project was in engineering the right prompts for the models which is a vital element of the framework. Specifically, I investigated a single prompt requesting three responses vs. separate prompts for each tone, role-playing prompting to condition the chatbot to respond as a human, designing the prompt to ensure that the generated response considers the limitations of MND and how the output should be formatted for downstream applications (see Listing B.3 for details). These prompts were also used later for the HuggingChat model.

Table A.1: Key Member Contributions

Key contributions of the members of the team

Contributor	Contribution
Wenxu Tian	GPT-3.5 Turbo model
Linh Nguyen	• GPT-3.5 Turbo model with memory • Introduced the concepts of LangChain and StreamLit • Developed user survey form
Soumya Ogoti	• Alpaca 7B model • Investigated prompt engineering techniques (tone, role-playing, templates) Individual contribution in this report: • Quantitative and qualitative analysis of survey results • Tone analysis using BERT • Fine-tuning of the GPT-3.5 Turbo model with memory
Naveen Dubey	HuggingChat model
Muhammad Sulaiman	BARD model

- I analysed the models with user survey results in detail both qualitatively and quantitatively (using Python libraries, Listing B.4).
- I analysed the tone of the responses output by the models using BERT (see Listing B.5). I created a custom dataset from the responses of the models for this purpose.
- Following the recent release of Fine-tuning API (22nd August 2023), I created a small dataset mimicking conversations of individuals with MND and fine-tuned the GPT-3.5 Turbo with memory model (Sec. B.4).

Appendix B

Code Snippets

Following are the code snippets for my contributions made in this report.

B.1 Alpaca 7B model and user interface

Code written for integrating the Alpaca 7B model with LangChain is presented in Listing B.1 and the code for the interface is presented in Listing B.2. Helper functions used for generating the prompts, configuring memory and parsing the responses are presented in Listing B.3.

```
from helpers import (
    get_chat_prompt_by_tone_alpaca,  # chat prompt
    get_memory,  # memory
    parse_response_multiple,  # parser
)

from langchain.chains import LLMChain
from langchain.llms import HuggingFacePipeline

from transformers import (
    LlamaForCausalLM,
    LlamaTokenizer,
    GenerationConfig,
    pipeline,
    BitsAndBytesConfig,
)

import time
```

```
19
20
21 def get_alpaca_llm():
22     quantization_config = BitsAndBytesConfig(
   llm_int8_enable_fp32_cpu_offload=True)
23
24     tokenizer = LlamaTokenizer.from_pretrained("chavinlo/
   alpaca-native", legacy=False)
25     alpaca_llm = LlamaForCausalLM.from_pretrained(
26         "chavinlo/alpaca-native",
27         load_in_8bit=True,
28         device_map="cpu",
29         quantization_config=quantization_config,
30     )
31
32     generation_config = GenerationConfig(
33         temperature=0.6,
34         quantization_config=quantization_config,
35         load_in_8bit=True,
36         top_p=0.95,
37         repetition_penalty=1.2,
38     )
39
40     pipe = pipeline(
41         "text-generation",
42         model=alpaca_llm,
43         tokenizer=tokenizer,
44         max_length=512,
45         generation_config=generation_config,
46     )
47
48     llm = HuggingFacePipeline(pipeline=pipe)
49
50     return llm
51
52
53 def alpaca_chains():
54     llm = get_alpaca_llm()
55
56     positive_chain = LLMChain(
57         prompt=get_chat_prompt_by_tone_alpaca("positive"),
58         llm=llm,
```

```
        memory=get_memory(llm),
        verbose=True,
    )

    neutral_chain = LLMChain(
        prompt=get_chat_prompt_by_tone_alpaca("neutral"),
        llm=llm,
        memory=get_memory(llm),
        verbose=True,
    )

    negative_chain = LLMChain(
        prompt=get_chat_prompt_by_tone_alpaca("negative"),
        llm=llm,
        memory=get_memory(llm),
        verbose=True,
    )

    chains = {
        "positive_chain": positive_chain,
        "neutral_chain": neutral_chain,
        "negative_chain": negative_chain,
    }
    return chains


def alpaca_predictor(alpaca_chains, text, chat_history):
    print("text ", text)
    print("chat_history ", chat_history)
    start_time = time.time()
    positive_response = alpaca_chains["positive_chain"].
   predict(input=text)
    t1 = time.time() - start_time
    print(positive_response)
    start_time = time.time()
    neutral_response = alpaca_chains["neutral_chain"].
   predict(input=text)
    t2 = time.time() - start_time
    print(neutral_response)
    start_time = time.time()
    negative_response = alpaca_chains["negative_chain"].
   predict(input=text)
```

```
    t3 = time.time() - start_time
    print(negative_response)

    print("avg time: ", (t1 + t2 + t3) / 3)
    return positive_response, neutral_response,
    negative_response


def alpaca_formatter(positive_response, neutral_response,
    negative_response):
    return parse_response_multiple(
        positive_response, neutral_response,
    negative_response
    )
```

Listing B.1: LLM chain with Alpaca 7B model

```
from alpaca_separate import (
    alpaca_chains,
    alpaca_formatter,
    alpaca_predictor,
)

from langchain.schema import AIMessage, HumanMessage

import streamlit as st


def set_chosen_response(input_state, chosen_response):
    st.session_state.state = input_state
    st.session_state.chosen_response = chosen_response


def initialize():
    st.title("Your personal assistant")

    if "state" not in st.session_state:
        st.session_state.state = "listening"
    if "chat_history" not in st.session_state:
        st.session_state.chat_history = []
    if "model_alpaca" not in st.session_state:
        st.session_state.model_alpaca = alpaca_chains()

```

```
27
28 def print_history(col):
29     with col:
30         # Print conversation history
31         st.subheader("Conversation History")
32         for message in st.session_state.chat_history:
33             st.write(message)
34
35
36 def main():
37     initialize()
38
39     # Split into two columns
40     col1, col2 = st.columns(2, gap="medium")
41
42     with col1:
43         # Receive message
44         st.session_state.received_message = st.text_input(
45             "Type the message here and click get suggestions
    button"
46         )
47
48         if st.button("Get suggestions") and st.session_state
   .received_message != "":
49             st.divider()
50             responses = alpaca_predictor(
51                 st.session_state.model_alpaca,
52                 st.session_state.received_message,
53                 st.session_state.chat_history,
54             )
55             formatted_response = alpaca_formatter(
56                 responses[0], responses[1], responses[2]
57             )
58
59             # Suggestion buttons
60             buttons = {
61                 tone: st.button(
62                     f"{tone}: {formatted_response[tone]}",
63                     on_click=set_chosen_response,
64                     args=("response_chosen",
   formatted_response[tone]),
65                 )
```

```
66                 for tone in ("Positive", "Neutral", "
   Negative")
67             }
68
69         if st.session_state.state == "response_chosen":
70             st.write(f"Your response: {st.session_state.
   chosen_response}")
71
72             # Update history with the chosen message
73             st.session_state.chat_history.append(
74                 "\nThem: " + st.session_state.
   received_message
75             )
76             st.session_state.chat_history.append(
77                 "\nYou: " + st.session_state.chosen_response
78             )
79
80             # Update memory of the models
81             st.session_state.model_alpaca["positive_chain"].
   memory.chat_memory.messages[
82                 -1
83             ] = AIMessage(content=st.session_state.
   chosen_response)
84             st.session_state.model_alpaca["neutral_chain"].
   memory.chat_memory.messages[
85                 -1
86             ] = AIMessage(content=st.session_state.
   chosen_response)
87             st.session_state.model_alpaca["negative_chain"].
   memory.chat_memory.messages[
88                 -1
89             ] = AIMessage(content=st.session_state.
   chosen_response)
90             st.session_state.state = "listening"
91         if st.button("Clear History"):
92             st.session_state.model_alpaca["positive_chain"].
   memory.clear()
93             st.session_state.model_alpaca["neutral_chain"].
   memory.clear()
94             st.session_state.model_alpaca["negative_chain"].
   memory.clear()
95
```

```
            st.session_state.state = "listening"
            st.write("Conversation history cleared!")

    print_history(col2)
```

Listing B.2: StreamLit interface for Alpaca 7B model

```
from langchain.prompts.chat import (
    ChatPromptTemplate,
    SystemMessagePromptTemplate,
    HumanMessagePromptTemplate,
)

from langchain.memory import ConversationBufferMemory,
    ConversationSummaryBufferMemory
import json


####################
## PROMPTS COMBINED
####################
def get_chat_prompt_combined():
    system_template = """ The following is a conversation
    between two people, a normal person A and a person with
    motor neuron disease B.
    The user giving the input is A and you are B. You should
    respond to A in a friendly manner.
    Keep in mind that B cannot do any physically intense
    activity.
    For every input you must provide one Positive, one
    Neutral and one Negative response formatted as a json
    object, with 3 keys: Positive, Neutral, Negative
    respectively.
    The value of each key should be a string, which is the
    response in that tone. The response must not be left
    empty.

    Current conversation:
    {history}
    """

    system_message_prompt = SystemMessagePromptTemplate.
    from_template(system_template)
```

```
26
27      human_template = "{input}"
28      human_message_prompt = HumanMessagePromptTemplate.
        from_template(human_template)
29
30      chat_prompt = ChatPromptTemplate.from_messages(
31          [system_message_prompt, human_message_prompt]
32      )
33
34      return chat_prompt
35
36
37  def get_chat_prompt_combined_2():
38      system_template = """The following is a friendly
        conversation between a person (A) and another person with
        Motor neuron disease (B).
39      B is talkative and provides lots of specific details
        from its context.
40      If B does not know the answer to a question, they
        truthfully say they do not know.
41      Your task is to suggest B 3 answers in 3 different tone
        of voices: Positive, Neutral, Negative.
42      Your suggestions need to be in a json object, with 3
        keys: Positive, Neutral, Negative.
43      The value of each key should be a string, which is the
        response in that tone. All 3 keys must always be present.
44  """
45      system_message_prompt = SystemMessagePromptTemplate.
        from_template(system_template)
46
47      human_template = """ Conversation:
48      {history}
49      A: {input}
50      B: """
51      human_message_prompt = HumanMessagePromptTemplate.
        from_template(human_template)
52
53      chat_prompt = ChatPromptTemplate.from_messages(
54          [system_message_prompt, human_message_prompt]
55      )
56
57      return chat_prompt
```

```
58
59
60 ####################
61 ## PROMPTS BY TONE
62 ####################
63 def get_chat_prompt_by_tone_alpaca(tone):
64     system_template = (
65         """ You follow instructions very well. The user
   inputs a conversation between two people, a normal person
    A and a person with motor neuron disease B.
66         Respond as if you are B in a single sentence only.
   The tone should always be """
67         + tone
68         + """. There should always be a response. B cannot
   do intense physical activity but do not mention this
   explicitly in the response."""
69     )
70
71     system_message_prompt = SystemMessagePromptTemplate.
   from_template(system_template)
72
73     human_template = """ Conversation:
74     {history}
75     A: {input}
76     B: """
77     human_message_prompt = HumanMessagePromptTemplate.
   from_template(human_template)
78
79     chat_prompt = ChatPromptTemplate.from_messages(
80         [system_message_prompt, human_message_prompt]
81     )
82
83     return chat_prompt
84
85
86 def get_chat_prompt_by_tone(tone):
87     system_template = (
88         """ The following is a conversation between two
   people, a normal person A and a person with motor neuron
   disease B.
89     For the input given by A respond as B in a """
90         + tone
```

```
        + """ tone.
    Keep in mind that B cannot do any physically intense activity.

    Current conversation:
    {history}
    """
    )

    system_message_prompt = SystemMessagePromptTemplate.from_template(system_template)

    human_template = "{input}"
    human_message_prompt = HumanMessagePromptTemplate.from_template(human_template)

    chat_prompt = ChatPromptTemplate.from_messages(
        [system_message_prompt, human_message_prompt]
    )

    return chat_prompt


def get_chat_prompt_by_tone_2(tone):
    system_template = (
        """ Read the conversation below and reply to the last message as if you are a patient with motor neuron disease in a """
        + tone
        + """ tone.
    Keep in mind that you cannot do any physically intense activity. Keep your response to a single sentence. Make sure not to mention about being a patient with MND.

    # Current conversation:
    # {history}
    # """
    )

    system_message_prompt = SystemMessagePromptTemplate.from_template(system_template)

```

```
125     human_template = "{input}"
126     human_message_prompt = HumanMessagePromptTemplate.
    from_template(human_template)
127 
128     chat_prompt = ChatPromptTemplate.from_messages(
129         [system_message_prompt, human_message_prompt]
130     )
131 
132     return chat_prompt
133 
134 
135 def get_chat_prompt_by_tone_3(tone):
136     chat_template = (
137         """ You follow instructions very well. The user
    inputs a conversation between two people, a normal person
     A and a person with motor neuron disease B.
138         Respond as if you are B in a single sentence only.
    The tone should always be """
139         + tone
140         + """. There should always be a response. B cannot
    do intense physical activity but do not mention this
    explicitly in the response."""
141         """ Conversation:
142     {history}
143     A: {input}
144     B: """
145     )
146 
147     chat_prompt = ChatPromptTemplate.from_template(template=
    chat_template)
148 
149     return chat_prompt
150 
151 def get_chat_prompt_by_tone_4(tone):
152     system_template = (
153         """ The following is a conversation between two
    people, a normal person A and a person with motor neuron
    disease B.
154     The user giving the input is A and you are B. You should
     respond to A in a friendly manner.
155     Keep in mind that B cannot do any physically intense
    activity.
```

```
156     You must provide a response in a """
157         + tone
158         + """ tone. The response must not be left empty.
    Make sure not to mention about being a patient with MND.
159
160
161     Current conversation:
162     {history}
163     """
164     )
165
166     system_message_prompt = SystemMessagePromptTemplate.
    from_template(system_template)
167
168     human_template = "{input}"
169     human_message_prompt = HumanMessagePromptTemplate.
    from_template(human_template)
170
171     chat_prompt = ChatPromptTemplate.from_messages(
172         [system_message_prompt, human_message_prompt]
173     )
174
175     return chat_prompt
176
177 ############
178 ## MEMORY
179 ############
180 def get_memory(llm):
181     memory = ConversationSummaryBufferMemory(
182         max_token_limit=100, llm=llm, ai_prefix="B: ",
    human_prefix="A: "
183     )
184     return memory
185
186
187 ###################
188 ## Parsers
189 ###################
190 def parse_response_from_json(response) -> dict:
191     """
192     Parses the response from a json object to a python
    dictionary
```

```
193     @param response: the input response
194     @return: the parsed response
195     """
196     parsed_response = json.loads(response)
197     return parsed_response
198 
199 
200 def parse_response_multiple(
201     positive_response, neutral_response, negative_response
202 ) -> dict:
203     """
204     Parses the response from three separate strings to a
    python dictionary
205     @param positive_response: the positive response
206     @param neutral_response: the neutral response
207     @param negative_response: the negative response
208     @return: the parsed response
209     """
210     parsed_response = {}
211     # if the output is in multiple lines, then only pick the
    first line
212     if "\n" in positive_response:
213         positive_response = positive_response.split("\n")[0]
214     if "\n" in neutral_response:
215         neutral_response = neutral_response.split("\n")[0]
216     if "\n" in negative_response:
217         negative_response = negative_response.split("\n")[0]
218 
219     # if there is a : in the response, then only pick the
    part after the :
220     if ":" in positive_response:
221         parsed_response["Positive"] = positive_response.
    split(": ")[1]
222     else:
223         parsed_response["Positive"] = positive_response
224 
225     if ":" in neutral_response:
226         parsed_response["Neutral"] = neutral_response.split(
    ": ")[1]
227     else:
228         parsed_response["Neutral"] = neutral_response
229 
```

```
230     if ":" in negative_response:
231         parsed_response["Negative"] = negative_response.
    split(": ")[1]
232     else:
233         parsed_response["Negative"] = negative_response
234 
235     return parsed_response
```

Listing B.3: Helper functions for prompts, memory and parsing

B.2 Analysis of user survey results

Code for generating visualizations and quantitative metrics from the user survey results is presented in Listing B.4

```
1 !pip install seaborn
2 !pip install openpyxl
3 # Import necessary packages
4 import numpy as np
5 import pandas as pd
6 import matplotlib.pyplot as plt
7 import seaborn as sns
8 
9 df=pd.read_excel("survey.xlsx")
10 
11 # Descriptive statistics of the dataset
12 
13 # Group by the chatbot
14 df.groupby('Chatbot').describe()
15 
16 # Check the distribution of overall ratings for each bot
17 # Get unique bot names
18 unique_bots = df["Chatbot"].unique()
19 print(unique_bots)
20 
21 # Set the number of bins (histogram bars)
22 num_bins = 10
23 
24 # Create subplots for each bot
25 num_plots = len(unique_bots)
26 fig, axes = plt.subplots(1, num_plots, figsize=(4*num_plots,
    8 ))
```

```
27
28 # Loop through each bot and plot its histogram in a separate
       subplot
29 for i, bot_name in enumerate(unique_bots):
30     bot_ratings = df[df["Chatbot"] == bot_name]["rating"]
31
32     axes[i].hist(bot_ratings, bins=num_bins, edgecolor="k",
       alpha=0.5)
33     axes[i].set_xlabel("Ratings")
34     axes[i].set_ylabel("Frequency")
35     axes[i].set_title(f"Chatbot Ratings Histogram - {
       bot_name}")
36     axes[i].set_xlim([0, 5])
37
38 # Adjust layout
39 plt.tight_layout()
40
41 # Save in eps format
42 plt.savefig('q1.eps', format='eps')
43
44 # Show the plot
45 plt.show()
46
47 df.groupby('Chatbot')['rating'].describe()
48 # get only chatbot and mean rating
49 df.groupby('Chatbot')['rating'].mean()
50
51 # Check the distribution of userfriendly ratings for each
       bot
52 import matplotlib.pyplot as plt
53 import seaborn as sns
54
55 # Set the style of seaborn
56 sns.set(style="whitegrid")
57
58 # Create a boxplot using Seaborn
59 plt.figure(figsize=(8, 4))
60 sns.boxplot(x="Chatbot", y="userfriendlyrating", data=df)
61
62 # Add labels and title
63 plt.xlabel("Chatbot")
64 plt.ylabel("User-friendliness Rating")
```

```
plt.title("User-friendliness Distribution by Chatbot")

# Rotate x-axis labels for better readability
# plt.xticks(rotation=45)

# explicitly specify the x-axis labels
plt.xticks([0, 1, 2, 3, 4], ["GPT-3.5 Turbo \n with memory",
    "Alpaca 7B", "BARD", "GPT 3.5 Turbo", "HuggingChat"], ha
   ="center")

# Save in eps format
plt.savefig('q2.eps', format='eps')

# Show the plot
plt.show()

df.groupby('Chatbot')['userfriendlyrating'].describe()

# Check the distribution of relevance ratings for each bot
import matplotlib.pyplot as plt
import seaborn as sns

# Set the style of seaborn
sns.set(style="whitegrid")

# Create a boxplot using Seaborn
plt.figure(figsize=(8,4))
sns.boxplot(x="Chatbot", y="relevancerating", data=df)

# Add labels and title
plt.xlabel("Chatbot")
plt.ylabel("Relevance Rating")
plt.title("Relevance Rating Distribution by Chatbot")

plt.xticks([0, 1, 2, 3, 4], ["GPT-3.5 Turbo \n with memory",
    "Alpaca 7B", "BARD", "GPT 3.5 Turbo", "HuggingChat"], ha
   ="center")

# Save in eps format
plt.savefig('q3.eps', format='eps')

# Show the plot
```

```
plt.show()

df.groupby('Chatbot')['relevancerating'].describe()

# Check the distribution of response time rating for each
    bot
import matplotlib.pyplot as plt
import seaborn as sns

# Set the style of seaborn
sns.set(style="whitegrid")

# Create a boxplot using Seaborn
plt.figure(figsize=(8,4))
sns.boxplot(x="Chatbot", y="responsetime", data=df)

# Add labels and title
plt.xlabel("Chatbot")
plt.ylabel("Response Time Rating")
plt.title("Response Time Rating Distribution by Chatbot")

plt.xticks([0, 1, 2, 3, 4], ["GPT-3.5 Turbo \n with memory",
     "Alpaca 7B", "BARD", "GPT 3.5 Turbo", "HuggingChat"], ha
    ="center")

# Save in eps format
plt.savefig('q4.eps', format='eps')

# Show the plot
plt.show()

df.groupby('Chatbot')['responsetime'].describe()

# check the Effectiveness Rating by chatbot
df.groupby('Chatbot')['effectiveness'].describe()
# check the percentage of each rating for each chatbot
effectiveness =df.groupby('Chatbot')['effectiveness'].
    value_counts(normalize=True) * 100
effectiveness

# plot effectiveness as bar chart for each chatbot
```

```
effectiveness.unstack().plot(kind='bar', figsize=(8, 4), rot=0)
plt.xlabel("Chatbot")
plt.ylabel("Percentage")
plt.title("Effectiveness of Resolving Issues by Chatbot")
plt.legend(loc='upper right')
plt.xticks([0, 1, 2, 3, 4], [ "Alpaca 7B", "BARD","GPT-3.5 Turbo \n with memory", "GPT 3.5 Turbo", "HuggingChat"], ha="center")

# Save in eps format
plt.savefig('q5.eps', format='eps')
plt.show()

# # Effectiveness Rating by chatbot in percentage
df.groupby('Chatbot')['effectiveness'].describe()
effectiveness =df.groupby('Chatbot')['effectiveness'].value_counts(normalize=True) * 100
effectiveness
```

Listing B.4: Code for analysing the user survey results

B.3 Tone Analysis with BERT

Code used for the tone analysis is presented in Listing B.5. Smart padding technique used for speeding up the training is adapted from (McCormick, 2020).

```
# !pip install transformers

# import libraries
import os
import torch
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import f1_score
from transformers import BertTokenizer, BertForSequenceClassification
from transformers import AutoConfig
from transformers import AutoModelForSequenceClassification
```

```
from transformers import AdamW
from transformers import get_linear_schedule_with_warmup
import matplotlib.pyplot as plt
from tqdm import tqdm
import time
import datetime
import random
import spacy
from torch.utils.data import TensorDataset, DataLoader

df1=pd.read_csv("generic_sentiment_dataset_50k.csv")
df2=pd.read_csv("generic_sentiment_dataset_10k.csv")

# combine the two datasets
df = pd.concat([df1, df2], ignore_index=True)
print(df['label'].value_counts())

# Take a subset of the data for faster processing
# 34500 rows of data from the original dataset
# 11500 rows for each of the 3 classes (positive, negative,
    neutral)
# set random seed for reproducibility

seed = 42
df_clean = df.dropna()  # drop rows with missing values
df_clean = df_clean.sample(frac=1).reset_index(drop=True)
           # shuffle the data
df_clean = df_clean.groupby('sentiment').head(11500)
     # take 11500 rows from each class
df_clean = df_clean.sample(frac=1).reset_index(drop=True)
     # shuffle the data again
df_clean

torch.cuda.empty_cache()

# Data splits
# -------------
# split the data into train, validation and test sets
seed1 = 42
seed2 = 52
X_train_and_val, X_test, y_train_and_val, y_test =
    train_test_split(df_clean.drop('label', axis=1), df_clean
```

```
    ['label'], test_size=0.2, random_state=seed1)
X_train, X_val, y_train, y_val = train_test_split(
    X_train_and_val, y_train_and_val, test_size=X_test.shape
    [0], random_state=seed2)

X_train_and_val_nn = X_train_and_val['text']
X_train_nn = X_train['text']
X_val_nn = X_val['text']
X_test_nn = X_test['text']

# check if there are any null values
print(X_train_and_val_nn.isnull().sum())
print(X_train_nn.isnull().sum())
print(X_val_nn.isnull().sum())
print(X_test_nn.isnull().sum())

### BERT Model (Optimized) - with Smart Padding
In order to improve over the previous model in terms of
    computatioal efficiency, smart padding technique is used
    where the padding is dynamically changed based on the
    length of the sequences.
# Load the BERT tokenizer.
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased
    ', do_lower_case=True)

tokens = []
labels = []
max_len = 256
update_interval = 1000

# Tokenize all training examples
print('Tokenizing {:,} training samples...'.format(len(
    X_train_nn)))

# For each training example...
for text in tqdm(X_train_nn):
    # Report progress.
    if ((len(tokens) % update_interval) == 0):
        print('  Tokenized {:,} samples.'.format(len(tokens)
    ))
    # Tokenize the sentence.
    input_ids = tokenizer.encode(text=text,
```

```
                                    add_special_tokens=True,
                                    max_length=max_len,
                                    truncation=True,
                                    padding=False)


    # Add the tokenized result to our list.
    tokens.append(input_ids)

print('DONE with {} samples'.format(len(tokens)))

# Sort training samples by the length of their input sequence.
train_samples = sorted(zip(tokens, y_train), key=lambda x: len(x[0]))
print('Shortest sample:', len(train_samples[0][0]))
print('Longest sample:', len(train_samples[-1][0]))

# Get the new list of lengths after sorting.
sorted_lengths = [len(s[0]) for s in train_samples]

# Plot the sequence lengths for visualization.
plt.plot(range(0, len(sorted_lengths)), sorted_lengths)
plt.xlabel('Sample Number')
plt.ylabel('Length of Sequence')
plt.title('Sequence lengths of training set samples')
plt.show()

def create_smart_padded_batches(text_samples, labels, batch_size, max_len = 256):
    print('Forming smart batches from {} examples. Batch size {}\n'.format(len(text_samples), batch_size))
    #
    ----------------------------------------------------------------------

    # First tokenize all the samples and truncate them to a max length.
    #
    ----------------------------------------------------------------------

    tokenized_inputs = []
    for sample in tqdm(text_samples):
```

```
116         # Tokenize the sample
117         tokenized_input = tokenizer.encode(text=sample,
118                                            add_special_tokens=True
    ,
119                                            max_length=max_len,
120                                            truncation=True,
121                                            padding=False)
122         # Add to list
123         tokenized_inputs.append(tokenized_input)
124     print('Finished tokenizing: {:>10,} samples\n'.format(
    len(tokenized_inputs)))
125
126     #
    ----------------------------------------------------------------------

127     # Batch the tokenized sequences together after sorting
    them by length.
128     #
    ----------------------------------------------------------------------

129     sorted_samples_labels = sorted(zip(tokenized_inputs,
    labels), key=lambda x: len(x[0]))
130
131     # Create lists of batches of samples and labels.
132     batch_ordered_sentences = []
133     batch_ordered_labels = []
134
135     # Loop over the input samples until it is empty
136     while len(sorted_samples_labels) > 0:
137         # Print progress after 500 batches.
138         if ((len(batch_ordered_sentences) % 500) == 0):
139             print(' Finished {:,} batches.'.format(len(
    batch_ordered_sentences)))
140
141         # num_samples_to_pick = normally batch size, for the
     last batch it will be the remaining samples
142         num_samples_to_pick = min(batch_size, len(
    sorted_samples_labels))
143
144         # Start at a random point in the list of samples and
     choose num_samples_to_pick contiguous samples
```

```
145         start = random.randint(0, len(sorted_samples_labels)
     - num_samples_to_pick)
146         batch = sorted_samples_labels[start:start +
    num_samples_to_pick]
147 
148         # Split the (samples, labels) tuples into samples
    and labels
149         batch_ordered_sentences.append([s[0] for s in batch
    ])
150         batch_ordered_labels.append([s[1] for s in batch])
151 
152         # Remove these samples from the original list of
    training samples.
153         del sorted_samples_labels[start:start +
    num_samples_to_pick]
154     print('\n  Finished processing {:,} batches.'.format(len
    (batch_ordered_sentences)))
155 
156     # --------------
157     #  Add Padding
158     # --------------
159     py_inputs = []
160     py_attn_masks = []
161     py_labels = []
162 
163     # For each batch...
164     for (batch_inputs, batch_labels) in zip(
    batch_ordered_sentences, batch_ordered_labels):
165         # Lists of padded inputs and masks
166         batch_padded_inputs = []
167         batch_attn_masks = []
168 
169         # Get the longest sequence size in the current batch
170         max_size = max([len(sen) for sen in batch_inputs])
171 
172         # Pad the sequences uptp the max_size and also
    create the attention masks.
173         for sen in batch_inputs:
174             padding_count = max_size - len(sen)
175             # Add the token for padding at the end of the
    sequence.
```

```
176             padded_input = sen + [tokenizer.pad_token_id]*
    padding_count
177             # Create the attention mask with 1 for real
    tokens and 0 for padding tokens.
178             attn_mask = [1] * len(sen) + [0] * padding_count
179             # Add these to the lists.
180             batch_padded_inputs.append(padded_input)
181             batch_attn_masks.append(attn_mask)
182 
183         # Convert to tensors and add to the lists.
184         py_inputs.append(torch.tensor(batch_padded_inputs))
185         py_attn_masks.append(torch.tensor(batch_attn_masks))
186         py_labels.append(torch.tensor(batch_labels))
187 
188     print('Finished Padding.')
189     token_count_padded = 0
190     for batch in py_inputs:
191         for sen in batch:
192             token_count_padded += len(sen)
193     token_count_fixed = max_len * len(py_inputs) *
    batch_size
194     print('  Number of tokens with fixed padding: {}'.format
    (token_count_fixed))
195     print('  Number of tokens with smart padding: {}'.format
    (token_count_padded))
196     print('  Percentage reduction in tokens is: {:,.2f}%'.
    format(100*(token_count_fixed-token_count_padded)/
    token_count_fixed))
197 
198     # Return the dataset
199     return (py_inputs, py_attn_masks, py_labels)
200 
201 def format_time(elapsed):
202     # Return time as string hh:mm:ss
203     time_int = int(round((elapsed)))
204     # Format as hh:mm:ss
205     return str(datetime.timedelta(seconds=time_int))
206 
207 # Create config and network for multiclass sequence
    classification.
208 config = AutoConfig.from_pretrained(
    pretrained_model_name_or_path='bert-base-uncased',
```

```
                                        num_labels=3)
model = AutoModelForSequenceClassification.from_pretrained(
    pretrained_model_name_or_path='bert-base-uncased',
    config=config)

print('Config:', str(type(config)), '\n')
print('Model type:', str(type(model)))

# Load the model on the GPU
device = torch.device('cpu')
if torch.cuda.is_available():
    device = torch.device('cuda')

desc = model.to(device)
print('DONE.')

# Define the optimizer
optimizer = torch.optim.AdamW(model.parameters(), lr = 2e-6,
    eps = 1e-8)

batch_size = 8
epochs = 4

# Max length for tokenization
max_len = 256

# Perform tokenization and smart batching on the training
   data
(py_inputs, py_attn_masks, py_labels) =
   create_smart_padded_batches(X_train_nn, y_train,
   batch_size, max_len)
total_steps = len(py_inputs) * epochs   # training_steps = [
   number of batches] x [number of epochs].

# Create the learning rate scheduler.
scheduler = get_linear_schedule_with_warmup(optimizer,
                                            num_warmup_steps
    = 0,

   num_training_steps = total_steps)

from functools import partial
```

```
244 from tqdm import tqdm
245 tqdm = partial(tqdm, position=0, leave=True)
246
247 # Set the seed value all over the place to make this
    reproducible.
248 seed_val = 87
249 random.seed(seed_val)
250 np.random.seed(seed_val)
251 torch.manual_seed(seed_val)
252 torch.cuda.manual_seed_all(seed_val)
253
254 training_stats = []
255
256 # Measure the total training time for the whole run.
257 total_t0 = time.time()
258
259 # For each epoch...
260 for epoch_i in tqdm(range(0, epochs)):
261     # ========================================
262     #               Training
263     # ========================================
264     print('\n=== Epoch {:} / {:} ==='.format(epoch_i + 1,
    epochs))
265
266     # Randomnize the training set, i.e smart batch again
267     if epoch_i > 0:
268         (py_inputs, py_attn_masks, py_labels) =
    create_smart_padded_batches(X_train_nn, y_train,
    batch_size, max_len)
269     print('Training on {:} batches...'.format(len(py_inputs)
    ))
270
271     start_time = time.time()
272
273     # Reset the total loss for this epoch.
274     total_train_loss = 0
275
276     # Put the model into training mode
277     model.train()
278
279     # For each batch of training data...
280     for step in tqdm(range(0, len(py_inputs))):
```

```
281         # Copy the current training batch to the GPU.
282         batch_input_ids = py_inputs[step].to(device)
283         batch_input_mask = py_attn_masks[step].to(device)
284         batch_labels = py_labels[step].to(device)
285 
286         # Clear gradients
287         model.zero_grad()
288         # Forward pass
289         result = model(
290             batch_input_ids,
291             token_type_ids=None,
292             attention_mask=batch_input_mask,
293             labels=batch_labels)
294 
295         # Compute training loss and accumulate
296         loss = result.loss
297         total_train_loss += loss.item()
298         # Backward pass
299         loss.backward()
300 
301         # To prevent exploding gradinets problem, clip the
    norm of the gradients to 1.0.
302         torch.nn.utils.clip_grad_norm_(model.parameters(),
    1.0)
303 
304         # Update the parameters and the learning rate.
305         optimizer.step()
306         scheduler.step()
307     # Calculate the average training loss over all of the
    batches.
308     avg_train_loss = total_train_loss / len(py_inputs)
309 
310     # Measure how long this epoch took.
311     training_time = format_time(time.time() - start_time)
312 
313     print("")
314     print("  Average training loss: {0:.2f}".format(
    avg_train_loss))
315     print("  Training epcoh took: {:}".format(training_time)
    )
316 
317     # ========================================
```

```
318     #                Validation
319     # ========================================
320     print("\nRunning Validation...")
321     (py_val_inputs, py_val_attn_masks, py_val_labels) =
    create_smart_padded_batches(X_val_nn, y_val, batch_size,
    max_len)
322
323     # Tracking variables
324     predictions , true_labels = [], []
325
326     # model.eval()
327     total_val_loss = 0
328     # For each batch of validation
329     for step in range(0, len(py_val_inputs)):
330         # Copy the batch to the GPU.
331         batch_input_ids_val = py_val_inputs[step].to(device)
332         batch_input_mask_val = py_val_attn_masks[step].to(
    device)
333         batch_labels_val = py_val_labels[step].to(device)
334
335         with torch.no_grad():
336             # Forward pass, calculate logit predictions
337             outputs = model(batch_input_ids_val,
    token_type_ids=None,
338                             attention_mask=
    batch_input_mask_val, labels=batch_labels_val)
339         total_val_loss += outputs.loss
340         logits = outputs.logits
341         # compute validation loss
342
343         # Move logits and labels to CPU
344         logits = logits.detach().cpu().numpy()
345         label_ids = batch_labels_val.to('cpu').numpy()
346
347         # Store predictions and true labels
348         predictions.append(logits)
349         true_labels.append(label_ids)
350
351     # Combine the results across the batches.
352     predictions = np.concatenate(predictions, axis=0)
353     true_labels = np.concatenate(true_labels, axis=0)
354
```

```
355     # Choose the label with the highest score as our
        prediction.
356     preds = np.argmax(predictions, axis=1).flatten()
357 
358     f1 = f1_score(true_labels, preds, average='macro')
359     # Calculate the average val loss over all of the batches
        .
360     avg_val_loss = total_val_loss / len(py_val_inputs)
361 
362     # Record all statistics from this epoch.
363     training_stats.append(
364         {
365             'epoch': epoch_i + 1,
366             'Training Loss': avg_train_loss,
367             'Training Time': training_time,
368             'validation loss': avg_val_loss,
369             'validation F1 score': f1
370         }
371     )
372 
373 print("")
374 print("Training complete!")
375 
376 print("Total training took {:} (h:mm:ss)".format(format_time
        (time.time()-total_t0)))
377 
378 training_stats
379 
380 # Evaluate on the test split
381 (py_test_inputs, py_test_attn_masks, py_test_labels) =
        create_smart_padded_batches(X_test_nn, y_test, batch_size
        )
382 
383 # Prediction on test set with the best model
384 print('Predicting labels for {:,} test batches ...'.format(
        len(py_test_labels)))
385 model.eval()
386 predictions , true_labels = [], []
387 
388 # For each batch of test
389 for step in range(0, len(py_test_inputs)):
390     # Copy the batch to the GPU.
```

```
    b_input_ids = py_test_inputs[step].to(device)
    b_input_mask = py_test_attn_masks[step].to(device)
    b_labels = py_test_labels[step].to(device)

    # Telling the model not to compute or store gradients,
   saving memory and
    # speeding up prediction
    with torch.no_grad():
        # Forward pass, calculate logit predictions
        outputs = model(b_input_ids, token_type_ids=None,
                        attention_mask=b_input_mask)

    logits = outputs[0]

    # Move logits and labels to CPU
    logits = logits.detach().cpu().numpy()
    label_ids = b_labels.to('cpu').numpy()

    # Store predictions and true labels
    predictions.append(logits)
    true_labels.append(label_ids)

print('    DONE.')

# Combine the results across the batches.
predictions = np.concatenate(predictions, axis=0)
true_labels = np.concatenate(true_labels, axis=0)

# Choose the label with the highest score as our prediction.
preds = np.argmax(predictions, axis=1).flatten()

# Calculate simple flat accuracy -- number correct over
   total number.
accuracy = (preds == true_labels).mean()

print('Accuracy: {:.3f}'.format(accuracy))
from sklearn.metrics import f1_score
f1 = f1_score(true_labels, preds, average='macro')
print('F1 score: {:.3f}'.format(f1))

# Save the model and tokenizer - This is the best model
model.save_pretrained("bert_model")
```

```
tokenizer.save_pretrained("bert_tokenizer")

### Final prediction
Predicting on the outputs of the different chatbot LLMs
Treating BERT predictions as ground truth labels and the
    tones predicted by the chatbot LLMs as the predicted
    labels.

torch.cuda.empty_cache()

# Load the model and tokenizer from the saved directory
model_name = "bert-base-uncased"
model = AutoModelForSequenceClassification.from_pretrained("
    bert_model/")
device = torch.device('cuda:0' if torch.cuda.is_available()
    else 'cpu')
model.to(device)  # Move the model to the same device as
    input tensors

# set model to evaluate mode
model.eval()

# Load the tokenizer from the saved directory
tokenizer = BertTokenizer.from_pretrained("bert_tokenizer/")

max_len = 256 # put that we chose during training

# test file generated from the models
test_data_files = ["tone_analysis_data_gpt.csv",
"tone_analysis_data_gptlc.csv",
"tone_analysis_data_alpaca.csv",
"tone_analysis_data_hc1.csv",
"tone_analysis_data_bard.csv"]

results_dict = {}

for test_file in test_data_files:
    chatbot_name = test_file.split('_')[3].split('.')[0]
    print('Predicting for chatbot: {}'.format(chatbot_name))

    df_test = pd.read_csv(test_file, encoding="cp1252")

```

```
468     # tokenize the reviews
469     test_encodings = tokenizer(list(df_test['text']),
    truncation=True, padding=True, max_length=max_len,
    return_tensors='pt', add_special_tokens=True)
470     print('Finished tokenizing: {:>10,} samples\n'.format(
    len(test_encodings)))
471
472     # create dataset loaders for ease of use
473     test_dataset = TensorDataset(test_encodings.input_ids,
    test_encodings.attention_mask)
474     test_loader = DataLoader(test_dataset, batch_size=16)
475
476     predictions = []
477     with torch.no_grad():
478         for batch in tqdm(test_loader):
479             # get a batch
480             input_ids, attention_mask = batch
481             input_ids, attention_mask = input_ids.to(device)
    , attention_mask.to(device)
482             # predict
483             outputs = model(input_ids, attention_mask=
    attention_mask, token_type_ids=None)
484             logits = outputs.logits
485             # store logits
486             predictions.append(logits.cpu())
487
488     # concatenate results from all batches
489     predictions = np.concatenate(predictions, axis=0)
490     # ground truth labels are argmax of logits as predicted
    by BERT
491     bert_labels = np.argmax(predictions, axis=1).flatten()
492
493     # create a dataframe with the predictions and the ground
     truth labels
494     results = pd.DataFrame()
495     results['model_labels'] = df_test['label']
496     results['bert_labels'] = bert_labels
497
498     results_dict[chatbot_name] = results
499
500     # confusion matrix
```

```
from sklearn.metrics import confusion_matrix, accuracy_score
    , f1_score

for chatbot_name, results in results_dict.items():
    print(chatbot_name)
    print('----------------')
    cmat = confusion_matrix(results['bert_labels'], results[
    'model_labels'])
    print("confusion_matrix:\n", cmat)
    print("accuracy: ", accuracy_score(results['bert_labels'
    ], results['model_labels']))
    print("f1 score: ", f1_score(results['bert_labels'],
    results['model_labels'], average='macro'))
    print('\n')
```

Listing B.5: Tone analysis with BERT

B.4 Fine-tuning GPT-3.5 Turbo

Code used for fine-tuning the GPT-3.5 Turbo model with memory on a custom dataset consists of four parts - checking the dataset and analysing the cost to perform fine-tuning (Listing B.6), uploading the dataset to OpenAI server (Listing B.7), creating a fine-tuning job on the OpenAI server (Listing B.8) and performing inference using the fine-tuned model (Listing B.9). These are adapted from (Witteveen, 2023).

```
from collections import defaultdict
import json
import tiktoken
import numpy as np


def check_for_errors(dataset):
    # Format error checks
    format_errors = defaultdict(int)

    for ex in dataset:
        if not isinstance(ex, dict):
            format_errors["data_type"] += 1
            continue

```

```
        messages = ex.get("messages", None)
        if not messages:
            format_errors["missing_messages_list"] += 1
            continue

        for message in messages:
            if "role" not in message or "content" not in message:
                format_errors["message_missing_key"] += 1

            if any(k not in ("role", "content", "name") for k in message):
                format_errors["message_unrecognized_key"] += 1

            if message.get("role", None) not in ("system", "user", "assistant"):
                format_errors["unrecognized_role"] += 1

            content = message.get("content", None)
            if not content or not isinstance(content, str):
                format_errors["missing_content"] += 1

        if not any(message.get("role", None) == "assistant" for message in messages):
            format_errors["example_missing_assistant_message"] += 1

    if format_errors:
        print("Found errors:")
        for k, v in format_errors.items():
            print(f"{k}: {v}")
    else:
        print("No errors found")


# Token counting functions
encoding = tiktoken.get_encoding("cl100k_base")


def num_tokens_from_messages(messages, tokens_per_message=3, tokens_per_name=1):
```

```
    num_tokens = 0
    for message in messages:
        num_tokens += tokens_per_message
        for key, value in message.items():
            num_tokens += len(encoding.encode(value))
            if key == "name":
                num_tokens += tokens_per_name
    num_tokens += 3
    return num_tokens


def num_assistant_tokens_from_messages(messages):
    num_tokens = 0
    for message in messages:
        if message["role"] == "assistant":
            num_tokens += len(encoding.encode(message["content"]))
    return num_tokens


def print_distribution(values, name):
    print(f"\n#### Distribution of {name}:")
    print(f"min / max: {min(values)}, {max(values)}")
    print(f"mean / median: {np.mean(values)}, {np.median(values)}")
    print(f"p5 / p95: {np.quantile(values, 0.1)}, {np.quantile(values, 0.9)}")


# load the dataset
dataset = []
with open("ft_valid_data.jsonl", "r") as f:
    for line in f:
        dataset.append(json.loads(line))

# check for errors
check_for_errors(dataset)

# Warnings and tokens counts
n_missing_system = 0
n_missing_user = 0
n_messages = []
```

```
convo_lens = []
assistant_message_lens = []

for ex in dataset:
    messages = ex["messages"]
    if not any(message["role"] == "system" for message in messages):
        n_missing_system += 1
    if not any(message["role"] == "user" for message in messages):
        n_missing_user += 1
    n_messages.append(len(messages))
    convo_lens.append(num_tokens_from_messages(messages))
    assistant_message_lens.append(num_assistant_tokens_from_messages(messages))

print("Num examples missing system message:", n_missing_system)
print("Num examples missing user message:", n_missing_user)
print_distribution(n_messages, "num_messages_per_example")
print_distribution(convo_lens, "num_total_tokens_per_example")
print_distribution(assistant_message_lens, "num_assistant_tokens_per_example")
n_too_long = sum(l > 4096 for l in convo_lens)
print(
    f"\n{n_too_long} examples may be over the 4096 token limit, they will be truncated during fine-tuning"
)

# Pricing and default n_epochs estimate
MAX_TOKENS_PER_EXAMPLE = 4096

TARGET_EPOCHS = 3
MIN_TARGET_EXAMPLES = 100
MAX_TARGET_EXAMPLES = 25000
MIN_DEFAULT_EPOCHS = 1
MAX_DEFAULT_EPOCHS = 25

n_epochs = TARGET_EPOCHS
n_train_examples = len(dataset)
print("Number of training examples: ", n_train_examples)
```

```
125
126 if n_train_examples * TARGET_EPOCHS < MIN_TARGET_EXAMPLES:
127     n_epochs = min(MAX_DEFAULT_EPOCHS, MIN_TARGET_EXAMPLES
    // n_train_examples)
128 elif n_train_examples * TARGET_EPOCHS > MAX_TARGET_EXAMPLES:
129     n_epochs = max(MIN_DEFAULT_EPOCHS, MAX_TARGET_EXAMPLES
    // n_train_examples)
130
131 n_billing_tokens_in_dataset = sum(
132     min(MAX_TOKENS_PER_EXAMPLE, length) for length in
    convo_lens
133 )
134 print(
135     f"Dataset has ~{n_billing_tokens_in_dataset} tokens that
     will be charged for during training"
136 )
137 print(f"By default, you'll train for {n_epochs} epochs on
    this dataset")
138 print(
139     f"By default, you'll be charged for ~{n_epochs *
    n_billing_tokens_in_dataset} tokens"
140 )
141
142 print("\n#### Pricing:")
143 PRICE_PER_THOUSAND_TOKENS = 0.008
144 print(
145     "Price at {} per 1000 tokens: ${:.2f}".format(
146         PRICE_PER_THOUSAND_TOKENS,
147         PRICE_PER_THOUSAND_TOKENS *
    n_billing_tokens_in_dataset / 1000,
148     )
149 )
```

Listing B.6: Validating the dataset for finetuning and computing cost

```
1 import openai
2
3 openai.api_key = "sk-
    aP7fDY8epzaxXW1TqrZPT3BlbkFJYYzT4xcMBAh7wh3gXB6A"
4
5 training_file_name = "ft_train_data.jsonl"
6 validation_file_name = "ft_valid_data.jsonl"
7
```

```
training_response = openai.File.create(
    file=open(training_file_name, "rb"), purpose="fine-tune"
)
training_file_id = training_response["id"]

validation_response = openai.File.create(
    file=open(validation_file_name, "rb"), purpose="fine-
    tune"
)
validation_file_id = validation_response["id"]

print("Training file id:", training_file_id)
print("Validation file id:", validation_file_id)
```

Listing B.7: Uploading dataset to OpenAI server

```
import openai

openai.api_key = 'sk-
    aP7fDY8epzaxXW1TqrZPT3BlbkFJYYzT4xcMBAh7wh3gXB6A'

training_file_id='file-G7g0OJhc7mas6CV8iswCMu8j'
validation_file_id='file-aA2mGNNAyHXKBTCKlPKbQKwW'

suffix_name = "arp-finetune"

# Create a fine-tuning job
response = openai.FineTuningJob.create(
    training_file=training_file_id,
    validation_file=validation_file_id,
    model="gpt-3.5-turbo",
    suffix=suffix_name,
)

job_id = response["id"]

print(response)

response = openai.FineTuningJob.retrieve(job_id)
print(response)

response = openai.FineTuningJob.list_events(id=job_id, limit
    =50)
```

```

events = response["data"]
events.reverse()

for event in events:
    print(event["message"])

response = openai.FineTuningJob.retrieve(job_id)
fine_tuned_model_id = response["fine_tuned_model"]

print(response)
print("\nFine-tuned model id:", fine_tuned_model_id)
```

Listing B.8: Submitting a fine-tuning job

```
from dotenv import find_dotenv, load_dotenv
from langchain.chains import ConversationChain
from langchain.chat_models import ChatOpenAI
from langchain.memory import ConversationBufferMemory,
    ConversationSummaryBufferMemory
from langchain.output_parsers import ResponseSchema,
    StructuredOutputParser
from langchain.prompts.chat import (
    ChatPromptTemplate,
    SystemMessagePromptTemplate,
    HumanMessagePromptTemplate,
)

load_dotenv(find_dotenv())


def get_memory(summary=False):
    memory_kwargs = {
        "human_prefix": "A",
        "ai_prefix": "B",
    }

    if summary:
        return ConversationSummaryBufferMemory(
    max_token_limit=100, **memory_kwargs)
    else:
        return ConversationBufferMemory(**memory_kwargs)

```

```
26
27 def get_conversation_chain(summary=False):
28     system_template = """
29     The user inputs a friendly conversation between two
   people, a normal person (A) and a person with motor
   neuron disease (B).
30     B is talkative and is capable of understanding context
   and remembering specific details from the conversation.
31     You are playing the role of B and must respond to the
   query of A with 3 answers in 3 different tones of voices:
    Positive, Neutral, Negative.
32     Make sure that you are aware of your limitations as a
   person with motor neuron disease but do not mention it
   explicitly in your resonses.
33     Your suggestions need to be in a json object, with 3
   keys: Positive, Neutral, Negative.
34     The value of each key should be a string, which is the
   response in that tone. All 3 keys must always be present.
35     {history}
36     """
37
38     system_message_prompt = SystemMessagePromptTemplate.
   from_template(system_template)
39
40     human_template = "{input}"
41     human_message_prompt = HumanMessagePromptTemplate.
   from_template(human_template)
42
43     chat_prompt = ChatPromptTemplate.from_messages(
44         [system_message_prompt, human_message_prompt]
45     )
46
47     chat_memory = get_memory(summary=summary)
48     chat_llm = ChatOpenAI(temperature=0.1, client="gpt-3.5-
   turbo")
49     # chat_llm = ChatOpenAI(temperature=0.1, client="ft:gpt
   -3.5-turbo-0613:bayes:arp-finetune:7sf3eo7j")
50
51     conversation = ConversationChain(
52         prompt=chat_prompt,
53         llm=chat_llm,
54         memory=chat_memory,
```

```
        verbose=True,
    )

    return conversation


class ChatOutputFormatter:
    def __init__(self):
        output_parsing_template = """
            Take the following text and reformat it. Do not change the text itself, only the format.

            text: {text}

            {format_instructions}
        """

        self.output_parser = StructuredOutputParser.from_response_schemas(
            [
                ResponseSchema(
                    name="Positive", description="Response in a positive tone"
                ),
                ResponseSchema(
                    name="Neutral", description="Response in a neutral tone"
                ),
                ResponseSchema(
                    name="Negative", description="Response in a negative tone"
                ),
            ]
        )
        self.format_instructions = self.output_parser.get_format_instructions()
        self.output_parser_ai = ChatOpenAI(
            temperature=0.0,
            client="gpt-3.5-turbo",
        )
        self.output_format_prompt = ChatPromptTemplate.from_template(
```

```
90              template=output_parsing_template,
91          )
92
93      def format_message(self, message):
94          message_format_prompt = self.output_format_prompt.
    format_messages(
95              text=message,
96              format_instructions=self.format_instructions,
97          )
98
99          formatted_message = self.output_parser.parse(
100             self.output_parser_ai(message_format_prompt).
    content,
101         )
102
103         return formatted_message
```

Listing B.9: Inference with fine-tuned model

Appendix C

Sample responses

Representative responses with the different models are shown in Figures C.1 C.2, C.3, C.4, C.5. Some of these responses were collated to create the dataset used for tone analysis.

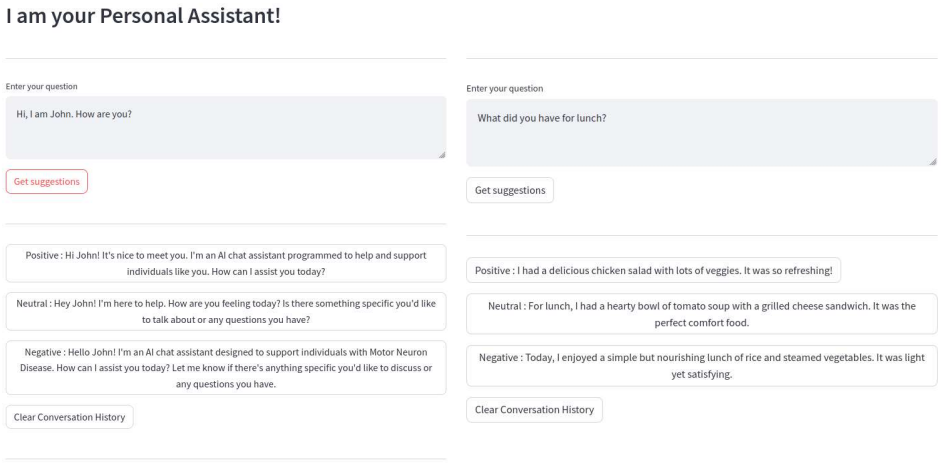


Figure C.1: Conversations from GPT-3.5 turbo

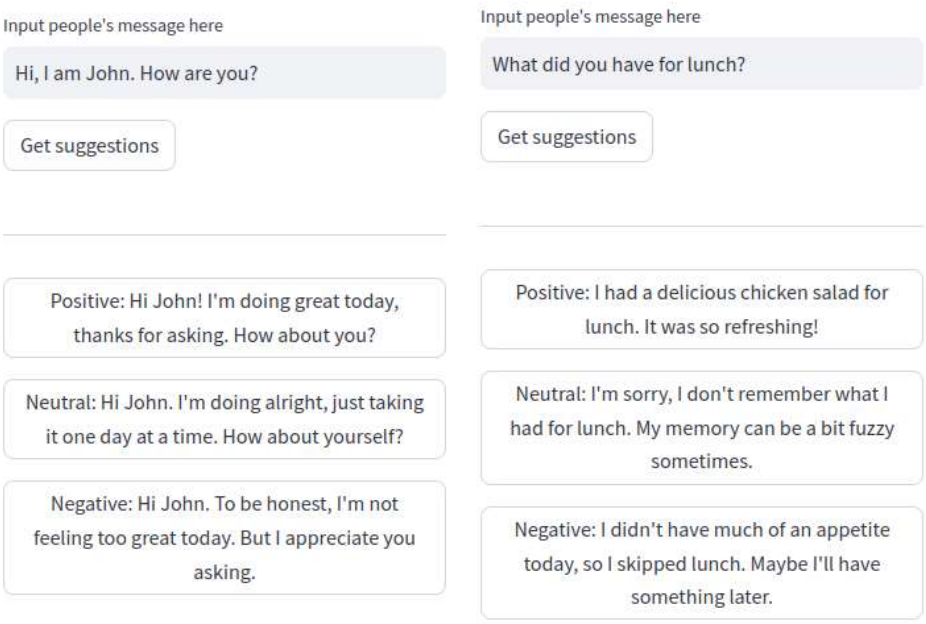


Figure C.2: Conversations from GPT-3.5 turbo with memory

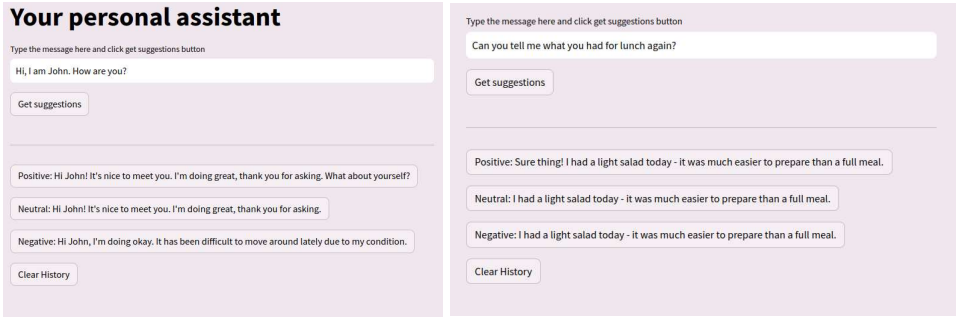


Figure C.3: Conversations from Alpaca 7B

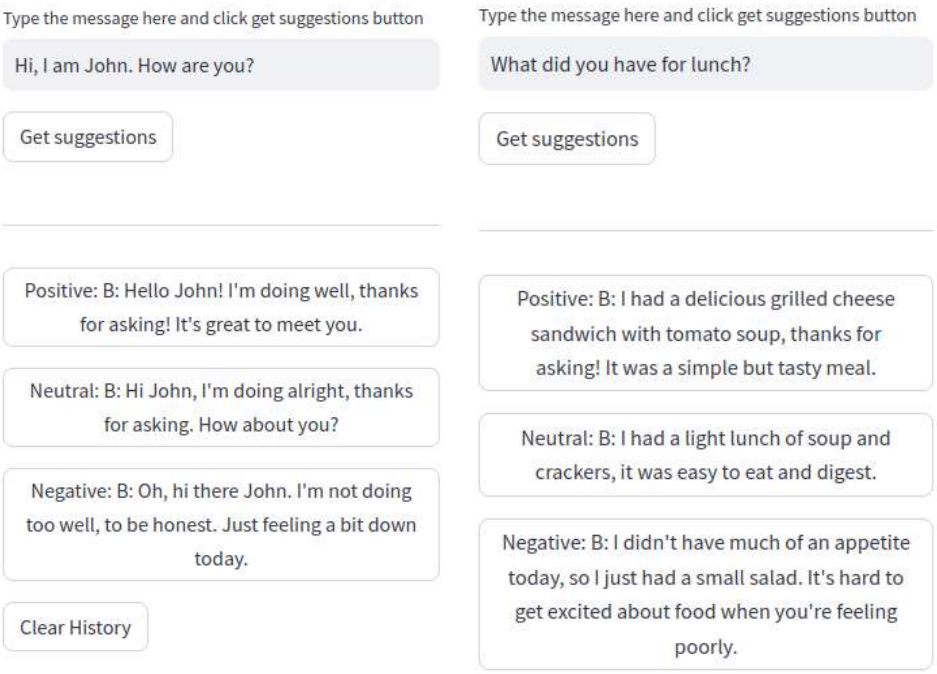


Figure C.4: Conversations from HuggingChat

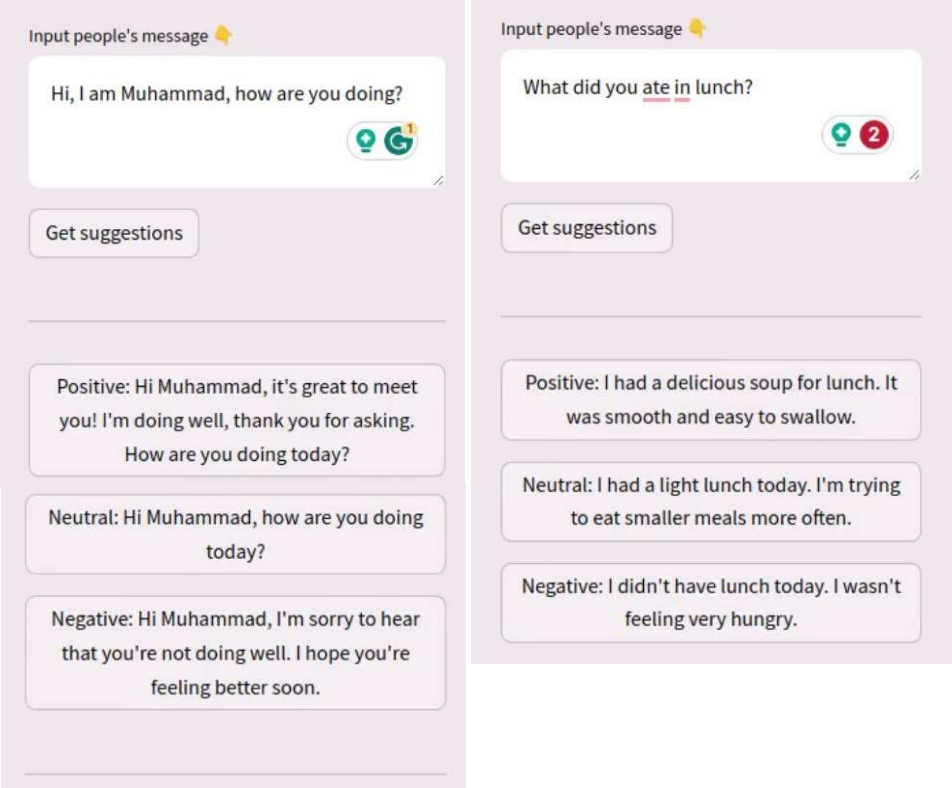


Figure C.5: Conversations from BARD

Bibliography

Anshu, 2021. Generic sentiment — multidomain sentiment dataset. https://www.kaggle.com/datasets/akgeni/generic-sentiment-multidomain-sentiment-dataset. [Online] Accessed: 2023-08-25.

Bahdanau, D., Cho, K., Bengio, Y., 2014. Neural machine translation by jointly learning to align and translate. arXiv preprint arXiv:1409.0473 .

Barker, A., 2018. Abilitynet remembers stephen hawking. https://www.abilitynet.org.uk/news-blogs/abilitynet-remembers-stephen-hawking-its-impossible-imagine-more-inspirational. [Online] Accessed: 2023-08-25.

Brown, T.B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D.M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., Amodei, D., 2020. Language models are few-shot learners. arXiv:2005.14165.

Cahalane, C., 2016. Effective communication, computing and tech for people with motor neurone disease (mnd). https://abilitynet.org.uk/news-blogs/effective-communication-computing-and-tech-people-motor-neurone-disease-mnd. [Online] Accessed: 2023-08-25.

Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H.W., Sutton, C., Gehrmann, S., Schuh, P., Shi, K., Tsvyashchenko, S., Maynez, J., Rao, A., Barnes, P., Tay, Y.,

Shazeer, N., Prabhakaran, V., Reif, E., Du, N., Hutchinson, B., Pope, R., Bradbury, J., Austin, J., Isard, M., Gur-Ari, G., Yin, P., Duke, T., Levskaya, A., Ghemawat, S., Dev, S., Michalewski, H., Garcia, X., Misra, V., Robinson, K., Fedus, L., Zhou, D., Ippolito, D., Luan, D., Lim, H., Zoph, B., Spiridonov, A., Sepassi, R., Dohan, D., Agrawal, S., Omernick, M., Dai, A.M., Pillai, T.S., Pellat, M., Lewkowycz, A., Moreira, E., Child, R., Polozov, O., Lee, K., Zhou, Z., Wang, X., Saeta, B., Diaz, M., Firat, O., Catasta, M., Wei, J., Meier-Hellstern, K., Eck, D., Dean, J., Petrov, S., Fiedel, N., 2022. Palm: Scaling language modeling with pathways. arXiv:2204.02311.

DAIR.AI, 2023. Prompt engineering guide. https://www.promptingguide.ai/introduction/settings. [Online] Accessed: 2023-08-25.

Devlin, J., Chang, M.W., Lee, K., Toutanova, K., 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv:1810.04805.

Hugging Face, 2023. Hugging face. https://huggingface.co. [Online] Accessed: 2023-08-31.

Intel, 2022. Intel helps give voice to people with mnd. https://www.intel.com/content/www/us/en/newsroom/news/intel-helps-give-voice-to-people-with-mnd.html. [Online] Accessed: 2023-08-31.

Jaspal'sVoice, . Allora communication device. http://www.jaspalsvoice.co.uk/pages/product/allora.jsp. [Online] Accessed: 2023-08-25.

Kohn, R., 2021. Mastering token limits and memory in chatgpt and other large language models. https://medium.com/@russkohn/mastering-ai-token-limits-and-memory-ce920630349a. [Online] Accessed: 2023-08-25.

Langchain, 2023. Langchain. https://docs.langchain.com/docs/. [Online] Accessed: 2023-08-25.

Liu, P., Yuan, W., Fu, J., Jiang, Z., Hayashi, H., Neubig, G., 2021. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. arXiv:2107.13586.

McCormick, C., 2020. Smart batching tutorial - speed up bert training. https://mccormickml.com/2020/07/29/smart-batching-tutorial/. [Online] Accessed: 2023-08-31.

Montague, J., 2017. The eyes have it: how technology allows you to speak when all you can do is blink. https://www.theguardian.com/society/2017/oct/16/eyes-have-it-motor-neurone-disease-technology-talk-blink. [Online] Accessed: 2023-08-25.

OpenAI, 2023. Models. https://platform.openai.com/docs/models/overview. [Online] Accessed: 2023-08-25.

Peng, A., Wu, M., Allard, J., Kilpatrick, L., Heidel, S., 2023. Gpt-3.5 turbo fine-tuning and api updates. https://openai.com/blog/gpt-3-5-turbo-fine-tuning-and-api-updates. [Online] Accessed: 2023-08-30.

Radford, A., Narasimhan, K., Salimans, T., Sutskever, I., et al., 2018. Improving language understanding by generative pre-training .

Raschka, S., 2023. Understanding large language models - a transformative reading list. https://sebastianraschka.com/blog/2023/llm-reading-list.html. [Online] Accessed: 2023-08-25.

Simone, S.D., 2023. Hugging face presents huggingchat, open source alternative to chatgpt. https://www.infoq.com/news/2023/04/hugging-chat-open-source/. [Online] Accessed: 2023-08-25.

Solaiman, I., Brundage, M., Clark, J., Askell, A., Herbert-Voss, A., Wu, J., Radford, A., Krueger, G., Kim, J.W., Kreps, S., McCain, M., Newhouse, A., Blazakis, J., McGuffie, K., Wang, J., 2019. Release strategies and the social impacts of language models. arXiv:1908.09203.

Taori, R., Gulrajani, I., Zhang, T., Dubois, Y., Li, X., Guestrin, C., Liang, P., Hashimoto, T.B., 2023. Alpaca: A strong, replicable instruction-

following model. Stanford Center for Research on Foundation Models. https://crfm.stanford.edu/2023/03/13/alpaca.html 3, 7.

Thoppilan, R., De Freitas, D., Hall, J., Shazeer, N., Kulshreshtha, A., Cheng, H.T., Jin, A., Bos, T., Baker, L., Du, Y., et al., 2022. Lamda: Language models for dialog applications. arXiv preprint arXiv:2201.08239 .

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., et al., 2023a. Llama: Open and efficient foundation language models. arXiv preprint arXiv:2302.13971 .

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., et al., 2023b. Llama 2: Open foundation and fine-tuned chat models. arXiv preprint arXiv:2307.09288 .

Treuille, A., Kelly, A., 2023. Generative ai and streamlit: A perfect match. https://blog.streamlit.io/generative-ai-and-streamlit-a-perfect-match/. [Online] Accessed: 2023-08-25.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I., 2017. Attention is all you need. Advances in neural information processing systems 30.

Wang, Y., Kordi, Y., Mishra, S., Liu, A., Smith, N.A., Khashabi, D., Hajishirzi, H., 2022. Self-instruct: Aligning language model with self generated instructions.

Witteveen, S., 2023. Fine tuning gpt-3.5 turbo. https://colab.research.google.com/drive/1ucLzDB_YJmyQe8DFEThqAUlN7NOi7UlP?usp=sharing&pli=1&authuser=1. [Online] Accessed: 2023-08-31.