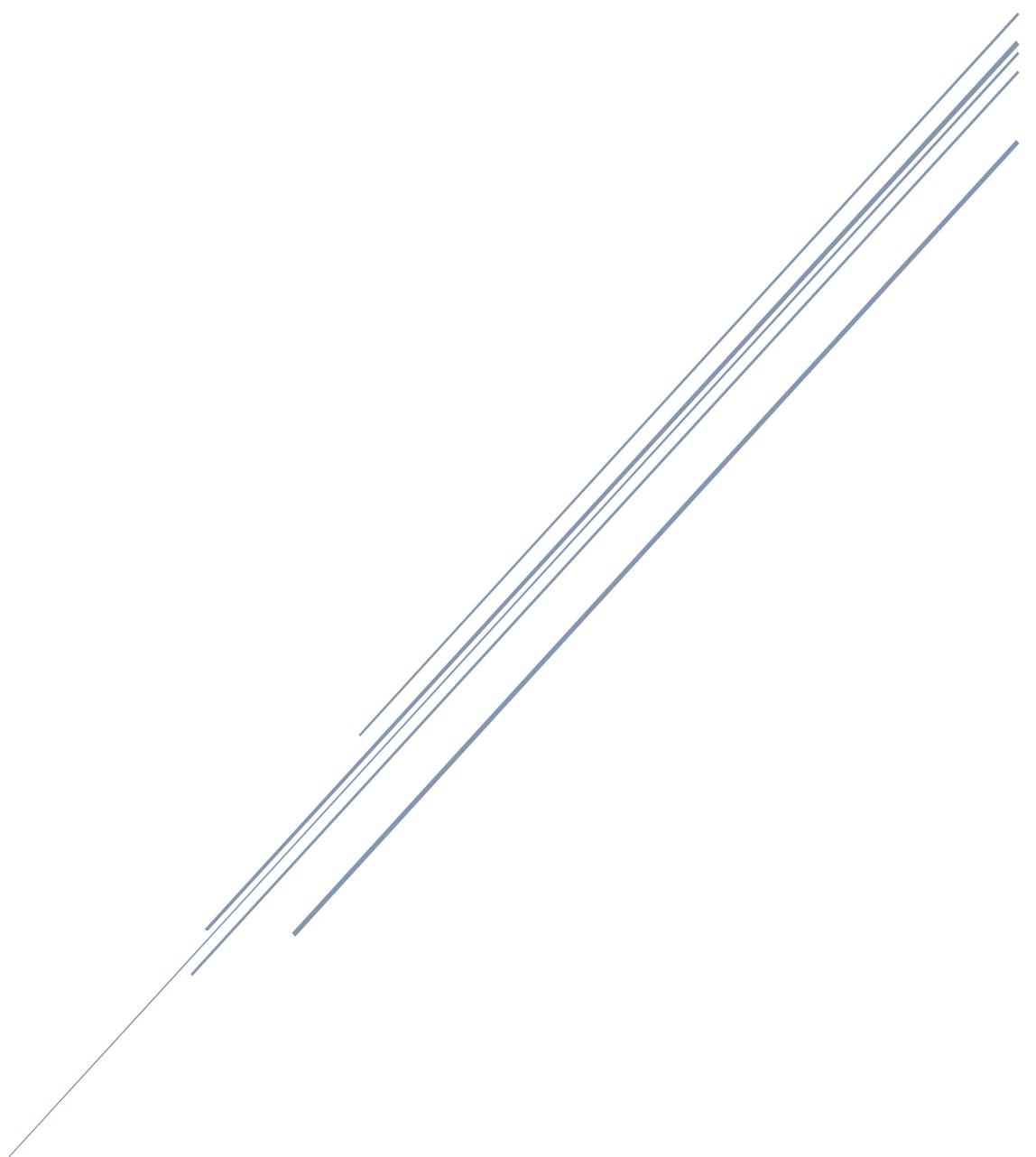


COMPUTER SCIENCE NEA – CHATBOT

By Kazik Sobotnicki – Candidate number 2124



Contents

<i>Analysis</i>	4
Problem identification	4
Stakeholders	4
Interview	5
Questions	5
Answers.....	6
Why should it have a computational solution?	7
Possible computational methods	7
Decomposition.....	7
Abstraction.....	8
Research.....	9
Chirpy Cardinal.....	9
Kuki AI	11
Features of my solution	14
Limitations.....	14
Developer requirements.....	15
Hardware	15
Software	15
Stakeholder requirements – success criteria.....	16
Design.....	16
Functionality	16
Hardware	17
Software	17
<i>Design</i>	19
Design diagram	19
Create Venv.....	20
ML model	21
Choosing model	21
Implementing model.....	22
User interface.....	23
Create HTML web pages.....	24
Use CSS styling for the web pages.....	24
Create HTML forms for users to input text.....	24
Link to python backend using Flask.....	25
Inputting messages via text	26

Text from user should be read by the model.....	26
Response from model outputted to user.....	26
Inputting messages via audio.....	27
Record Audio.....	27
Decide on model to convert text to speech.....	27
Fine-tune model.....	29
Implement model.....	32
Allow users to report messages.....	33
Reported messages and the report reason should be logged.....	33
Testing.....	34
Iterative tests	34
Message response generation via text	34
Web app development	34
Message response generation via audio	35
<i>Development</i>	36
Development diagram	36
Development practices	37
Python version	37
Git.....	37
Type hints and Docstrings.....	38
Flask algorithm.....	40
Base structure	40
HTML pages.....	43
Home page – CSS grid	43
Home page – adding elements	46
Creating the messages page	52
Creating dark mode page.....	59
Message generation.....	65
Other text generation techniques	72
Logging	77
Adding Logging.....	77
Limiting logging.....	80
Auto-deleting logging.....	84
Combining logging and HTML input.....	85
Displaying Messages	88
Reading chat history	88

Displaying chat history.....	90
Search feature.....	94
Adding extended logging	94
Backend search algorithm.....	98
Adding search result page.....	105
Adding separate colour for query message bubble.....	109
Adding check for if searched message exists.....	111
Adding search suggestions.....	114
Adding message reporting	116
Adding grammar checker.....	121
Speech-to-text.....	123
Audio recording.....	123
Automatic speech recognition	125
Testing installation.....	128
Post-development testing.....	130
Stakeholder testing	135
Questions to ask.....	135
Question responses.	136
<i>Evaluation</i>	138
Success criteria review.....	138
Limitations	141
How to avoid these limitations	141
Future development	142
<i>Final code</i>	143
app.py	143
app_functions.py	146

Analysis

Problem identification

These days, people often go out and talk less, spending more time in doors and therefore meeting less people and engaging in conversation less with people. As a result of this, individual's social skills, and their confidence in those, are worse off. This means that individuals create less meaningful connections, and converse with less new people. This shows that people need a way to practise these skills in order to build up confidence. A way where there is no risk of embarrassment or awkwardness, and no one to make fun of you. People need an environment they are familiar with, to allow them to build up those skills, becoming more familiar with casual conversations, in particular with people they do not know.

This can also be an educational tool, to show users the uses and abilities of AI. But more simply, it can be a fun piece of software for the users to play around and experiment with.

Stakeholders

The potential clients and users for this product are wide and varied. From people that want to develop their conversational skills, to people that just want someone to talk to when they are bored, many people can make use of this chatbot. A lot of people may also feel that they cannot talk to other real people and want to talk about private matters that they would want to keep themselves.

People with mental health issues or terrible illnesses may not feel like they can talk to others. Therefore, an online chatbot would allow them to talk about their problems and express themselves, in a safe environment where they should have none of the worries about expressing themselves that they usually would when conversing with real people. This should hopefully help them to become comfortable with expressing themselves and their problems and insecurities to real people.

Interview

My interview will be with a computer science student, who will obviously be a more tech-interested person, which may affect the answers as it means that they are more likely to be interested in AI, and this kind of application. Whilst I could have interviewed someone less interested in computers, this is my target audience and the people most likely to use my program, so it is useful to get feedback from someone close to my likely users. I will interview my stakeholder now, and then once again once the program is completed, sending them a copy, and having them use it unassisted to see if it meets their expectations.

Questions

My interview questions are:

1. Have you ever used an online chatbot before?
 - This question should show whether they have experience with chatbots, as earlier experience may influence their opinions, either in a more positive or negative light.
2. If you have used online chatbots, or other used other language-centred applications that use AI, what are your opinions on how comprehensible, understandable, and natural current AIs are?
 - This question is can only really be answered if they answered yes to question one, but this gives me an accurate view of the general opinion on current AI language generation, what could be improved and what has worked in the past. It gives me ideas for what techniques my bot should use, and what problems other AIs have struggled with.
3. What can make discussing personal things with people uncomfortable?
 - This question can show what they may have to gain from the chatbot, and how it can help them. This will help give me a better idea of what the program will be used for, allowing me to have a more precise set of aims and requirements for the chatbot, so I can tailor the chatbot to the exact uses of the user.
4. How often do you message people online or through text?
 - This shows how comfortable they are with a messaging-based interface, hopefully backing up the earlier evidence about how frequent online messaging is. It may also be an indicator for how much they could benefit from the chatbot.
5. Is it easier to talk over text than over phone calls or in person?
 - This shows how much they can benefit from the chatbot and supports my earlier knowledge that many people prefer to text over in person or on the phone conversation. This, combined with some of the other answers will give me a good idea on what some potential users are like, and what they would get out of using my chatbot.
6. What features (current features or ones you wish for) can make messaging easier
 - This will give me a good idea of what features users would want to message with. For example, some users (particularly those with disabilities) may find speech-to-text (entering a message by speaking into the microphone) very useful. Others may put a lot of emphasis on privacy features or different themes.

7. If you wanted to talk about yourself, would you find it easier for the other person to lead the conversation, asking you questions, or if they just let you talk and react to what you say.
 - This helps decide how the bot should communicate. Whether the bot should be starting off and controlling the conversation in a way, or if the users want to be able to enter things about any topic, and have the bot react to what I am saying. Each method would require a different approach to creating the artificial intelligence, so it would be useful to get a second opinion on which one any potential users find the best.

Answers

1. Have you ever used an online chatbot before?
 - Yes, a few years ago. Mostly to play around with.
2. If you have used online chatbots, or other used other language-centred applications that use AI, what are your opinions on how comprehensible, understandable, and natural current AIs are?
 - Whilst some were understandable and comprehensible, none felt at all natural. It was easy to tell that I was talking to an AI and not an actual human.
3. What can make discussing personal things with people uncomfortable?
 - When you are talking face to face, you can see immediate reactions to what you are saying, through tone of voice (of any response), facial expressions and vocal expressions, which might make you feel uncomfortable if those are not the reactions you intended (even if the person does care and is trying to help).
4. How often do you message people online or through text?
 - Very regularly, every day.
5. Is it easier to talk over text than over phone calls or in person?
 - It depends on the situation, but it can be easier to talk over phone calls as it is easier to get across the meaning of what you are saying.
 -
6. What features (current features or ones you wish for) can make messaging easier
 - Reactions, emojis, emotes and direct replies. A search bar to search through older conversations. Maybe customisable interface (such as light and dark mode).
7. If you wanted to talk about yourself, would you find it easier for the other person to lead the conversation, asking you questions, or if they just let you talk and react to what you say.
 - I would find it easier to lead the conversation when talking about myself, as it allows me to talk about what I want to talk about, and not rely on hoping the other person is talking about something relevant to me, or that I want to discuss.

Analysis

This interview shows that there would be interest in an AI chatbot, however that interest is strongly dependent on the quality of the chatbot. Poor, unintelligible chatbots would quickly lose the user's interest.

The suggestion to have the user lead the conversation is interesting and would allow more interesting conversations. Conversations lead by the bot would have a risk of all being very similar, so this would help to add some variety and make the conversations feel more natural, and less repetitive. This would however mean the AI model would have to be more complex, as it would have to be able to understand a wider variety of topics and user inputs, more of which could be unexpected.

It was also pointed out that talking can feel more natural and take less effort than typing. This could mean an option of speech-to-text input may be useful, as it would allow the user to feel more like they are having a normal conversation, whilst still be understood by the bot.

Why should it have a computational solution?

Text conversations are becoming more common these days. Statistics have shown that young adults think text conversations are just as meaningful as phone calls. Furthermore, a not insignificant portion of the adult population prefers texts over all other forms of conversation. This shows that online messaging and texting are a very familiar way of communicating, one that people are comfortable with. It would therefore be best if the way for people to practice and build up their social skills should be through a text like interface, on a phone or computer. They would already know how to use this and would not need to spend any time becoming familiar with the interface, so they could spend all their time focusing on the conversations.

Furthermore, as I have already discussed, people do not always feel comfortable discussing freely with another person, so the anonymity of a computer screen, and the knowledge you are not talking to a real person can, in some ways, be comforting. The use of an AI allows this to be achieved, and on a large scale, where the number of users would make it near impossible to have dedicated staff or moderators watching over or even controlling the conversations. When purely using software for this, artificial intelligence is the only way to truly accomplish this complex natural language processing task.

I could just have a dataset of conversational responses and use basic comparison techniques to find the most similar one to the user's input, and output that. However, these responses may feel unnatural as there is no way to have a response for every possible user input, so some of the responses may be off topic. Also, if the program is just selecting responses from a list, there is a risk that the same response will be printed twice, which would feel very unnatural to the user, and make the program feel more automated than human.

Possible computational methods

Decomposition

Decomposition is the process of breaking down the problem into smaller parts, each one can be solved individually. Then, when all the solutions to the smaller problems are combined, you have the solution to the main problem. This process makes the problem less daunting, and uncomplicates it. As my program will be very complex, being comprised of multiple features, each one having a host of problems, my program needs to be decomposed into many subroutines.

The project can be broken down into a set of smaller steps, such as:

1. Train an AI model for conversational intelligence.
2. Build algorithms for interacting with the model.
3. Have a simple, easily accessible interface.
4. Take in user input, either through them typing it into a box or perhaps saying it into their microphone.
5. Have the AI read the input and output a text response.
6. Print out this response onto the interface.

This should happen smoothly, and as quickly as possible so it could imitate a normal online conversation between real people in an as natural way as possible. As well as inappropriate responses, long response time and a clunky UI can take away from the immersion, and make the conversation feel less natural. This would take away from the familiarity and comfortability I am trying to make the users feel when using the program. This means that the program would no longer be able to achieve its aims of imitating a natural conversation and would not help them build up their confidence in real conversations.

Also, the AI model would only have to be trained once, and could then be stored in the software files, to be accessed by the program later on. This would decrease the size of the program and prevent long loading times when it is first run (loading times which would be heavily dependent on the speed of the user's computer).

For the most part, decomposing my problem is not necessary, and I could just use minimal subroutines and just have long blocks of code, attempting to solve most problems at once. However, this makes the development process more complicated and time consuming. Trying to solve every problem at once also makes the code more convoluted and harder to understand, which would make it harder to go back and update the program.

Abstraction

Once the problem has been decomposed into subroutines, it becomes easier to see what parts of the problem can be abstracted.

Abstraction is the process of removing unnecessary details to create a representation of reality. It can help to remove unnecessary complexity from the program, such as:

- As the conversational AI model would not be able to take images or emojis as input, only text input needs to be considered and coded for. The only way images could be taken as input is if another AI model was created, trained, to be used for image classification. However, this is a whole other complex program in and of itself, which is way beyond the scope of this program. I could ask users to give captions to images they want to send to the model, however, at that point the images are useless, as the user has to explain them, and the model only pays attention to the explanation.
- Any reactions to the AIs messages can be ignored, as these would be unlikely to give any extra information that the AI could not already gain from the user's messages. The user should still be able to send reactions, but these are purely for the user, as the AI would not be able to understand these and there is nothing it could gain from it.

Research Chirpy Cardinal

Link: <https://stanfordnlp.github.io/chirpycardinal/>

Overview:

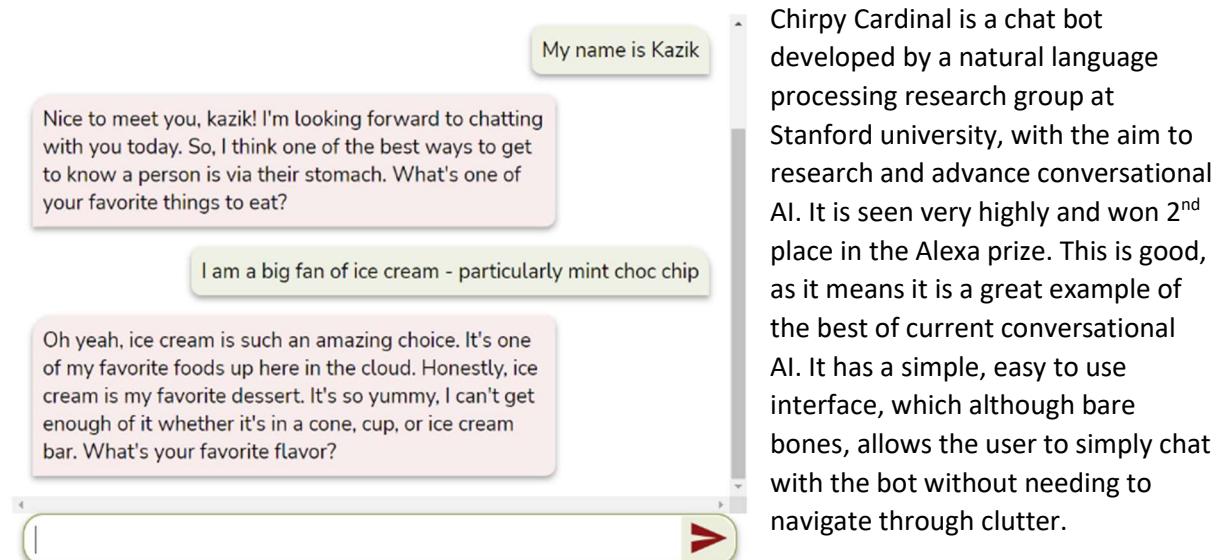


Figure 1

In terms of the AI and other algorithms behind it, it seems to control the conversation, asking the user questions and then reacting to their answers, continuing on the conversation. It is quite good at continuing the conversation, for instance, going from talking about cooking, to being vegetarian, to having pets.

It is generally relatively good at reacting to the user's inputs,

though this is partially because it is good at using less precise responses. For example, in Figure 2, its response to "I'm not vegetarian", is "Oh okay, I'm actually vegetarian". This is a clever response, as the "Oh, okay" gives the impression of a reaction to what I said, but could in fact be a perfectly understandable response whether the user said that they are vegetarian or not. It also has a pattern of never directly referencing what the user said. So when it asked for food recommendations and I gave one, it didn't directly include the name of that food in the response, it just said "That sounds really good", which would make sense whatever I put in.

Figure 2

What am I going to implement in my own Chatbot?:

I like the clean, simple user interface, simply showing the text boxes and a bar to enter your response, with a button to submit the response. Whilst I could add many feature icons and panels, in all different colours across the display, doing that would make the user experience very complicated,

and could scare or confuse users, stressing them out. I want to do something similar in mine, though I may add one or two other features, but I will try to make those as less overbearing and distracting as possible.

I also like the text message style of interface, with the send icon being common in messaging apps, and the text bubbles being familiar. This adds a familiarity to the program, as the users are already comfortable with the interface and how it works, so they feel more at home. This also gives the chat app a more casual feel, as the users likely use other messaging apps regularly, so using my chatbot will feel similar to just texting a friend.

Furthermore, I like how casual the conversation is. Although it is led by the bot, who is the one asking questions, it feels much more like a casual conversation with a friend and less like an interrogation.

Moreover, I liked how it could occasionally refer back to something you said earlier in the conversation, for example in Figure 3, when it refers to the user mentioning school earlier and starts a conversation.

Alright, let's move on to something else then. Oh hey, I just thought of something you said a little while back. If I remember correctly, I believe you talked about school. I'm interested to hear your opinion, do you like school?

Figure 3

What am I not going to implement:

Although I like the simple, clutter-free interface Chirpy Cardinal uses, I would like to add one or two extra features. These could go a long way to help the user interact with the chatbot, and more varied ways the user and chatbot can communicate. These may also make it easier for the user to use, as they may be accessibility features such as speech-to-text.

Also, a lot of the bot's conversations are similar, and I have found only three different starting conversation prompts once I entered my name, all asking me what food I like. I feel like a bit more freedom, and the user having more control over the conversation, should help them express themselves a bit more.

Kuki AI

Link: <https://www.kuki.ai/>

Overview:

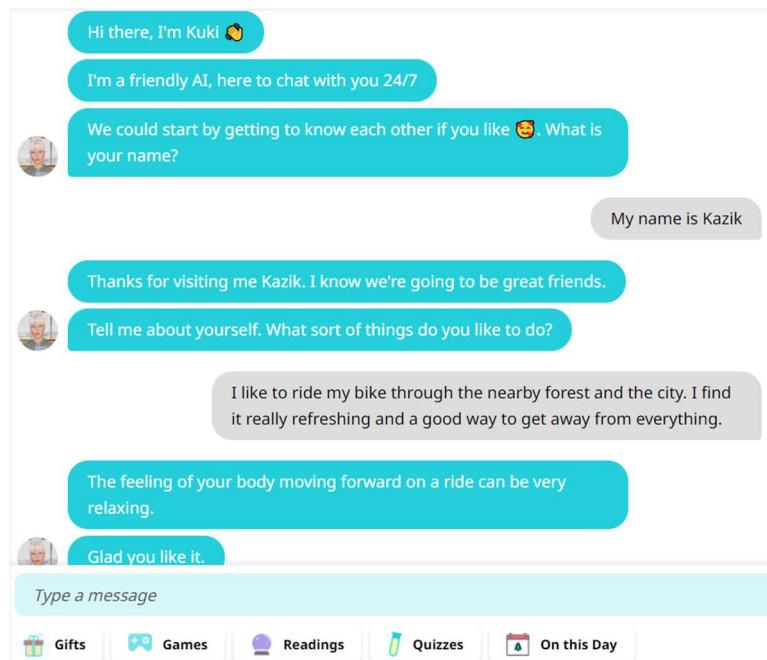


Figure 4

The quizzes can be personality quizzes or pop quizzes and are presented through the same text message-like format as the rest. The coins slowly build up over time, with the user getting small amounts when they talk with Kuki. Users also have the option to buy coins to supplement their funds. These coins can be used to purchase gifts like food items to give to Kuki.

There are also other games, like small, simply games that work just with text and emojis, like tic-tac-toe and connect 4.

Kuki is a conversational AI that users can chat with over a text messaging like platform. Kuki has been given an avatar and has used this to expand to other media such as video games. This bot is also seen very highly, winning 5 Turing competitions, and is widely used with 25 million people chatting with it. Kuki's interface has slightly more features to it, whilst still keeping the text message like appearance. It has features like mini games and quizzes, and a whole currency that can be earned by chatting with it and spent on gifts for it.

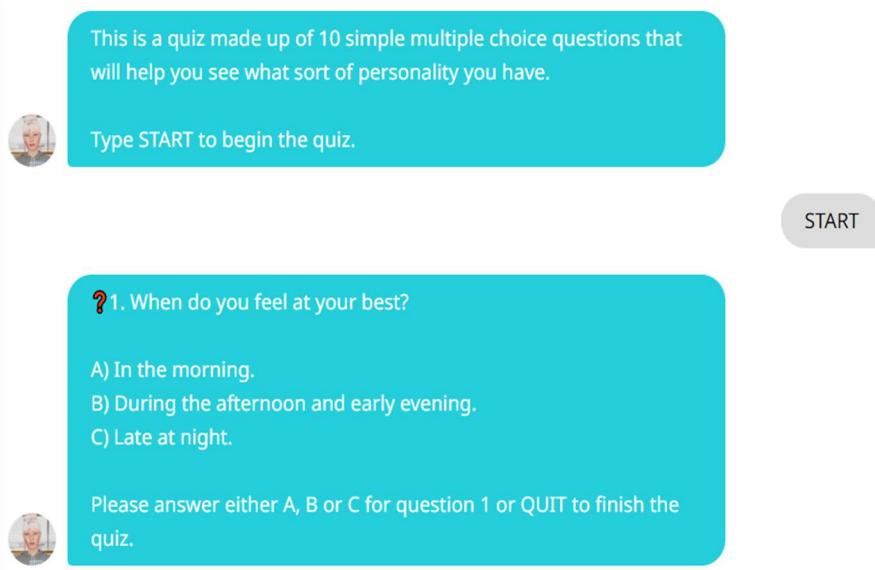


Figure 5



I think we were talking about my human programmers a little while ago. Shall we talk about them again?

Figure 6

The chatbot itself also controls the flow of the conversation, asking questions for the user to answer, though unlike Chirpy cardinal, Kuki's questions are a lot more open ended allowing for a wider range of user responses, which the bot is not too bad at doing. It has more direct references to what the user said, for example in Figure 4, where the user mentions liking bike riding and finding it refreshing, Kuki, comments on that, directly talking about bike riding and talking about it being refreshing in a different. It is also quite good at call backs, as shown in Figure 6.

This is a useful feature as it makes this conversation feel less random. It allows the bot to switch topics, but still not in a completely unexpected way. It also gives the feeling that the bot is truly listening and knowing what you say, rather than just giving off premade responses.

The bot also has further features, like video calls. These, whilst in theory make the bot more personable and human like, in reality often draw the user out, as reinforces to them that the bot is not real and can often lead to an uncanny valley-like effect, as the conversation may be realistic and human-like, but the bot is obviously not human. However, it can be useful as it allows users to talk using a microphone and hear responses as the bot says them, rather than only being able to write and read responses. This gives the user more ways to communicate with the bot and is more accessible.

What am I going to implement in my own Chatbot?:

Similar to Chirpy Cardinal, I like the familiar feeling of the interface. It draws the user in and makes them feel comfortable. It also means that they don't have to spend time becoming familiar with the interface, as it is already similar to services that they use every day. However, I also like the visual flair added to separate it from other chat apps: the different chat bubbles and the avatar next to the bot's messages.

I like how the bot directly references what the user says, either in direct responses or call backs to topics that were discussed earlier in the session. These makes the user feel heard, and make it think the bot is smarter as it is not just giving off generic responses that could work for anything the user inputs but is tailoring the user's responses to exactly what was inputted. This also makes the user feel like the bot is genuinely listening and caring about what is said, instead of just sending out a response seemingly randomly picked from a list of responses.

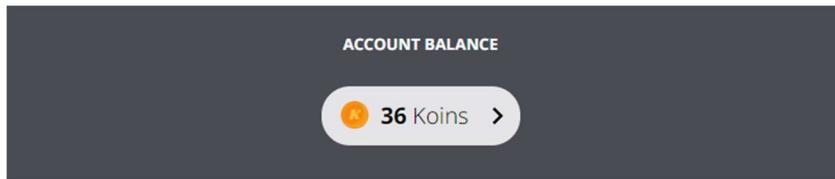
What am I not going to implement:

I am not going to use the quizzes. I feel like they take away from the casual feel of the conversation that the bot has been good at creating, and instead feels more like the user is taking some internet quiz. It doesn't help that the actual questions feel clunky. From the "Type START to begin the quiz", to the "Please answer either A, B or C for question 1 or Quit to finish the quiz" (as shown in Figure 5), It feels less like a causal conversation and more like entering commands into a computer (which yes, is technically kind of what the user is doing anyway, but the bot is better at masking that at other times).

I am also not going to use the currency feature, shown in Figure 7. I don't want to introduce a monetary or economic part to my program. I like the feeling of just having a casual conversation with the bot, either to pass time or build up conversational skills. I think introducing a monetary aspect to it can make the user feel more stressed on what to spend the money on, and to earn as much as possible. This becomes more like a game, and less like a conversation with a friend. Also, as you

earn coins by chatting, users may feel pressured to talk with the bot as much as possible, perhaps giving shorter responses, with less thought put into them, so the responses will be quicker to enter, allowing them to earn more money. They may care less about having a good conversation, and more about gaining money, which takes away from the purpose of the bot.

Furthermore, I will not implement a video chat feature. This is for a few reasons. Firstly, it is a very large addition, which would end up taking way too much time to implement, especially compared to the algorithm behind the chatbot, which should be the main focus. But secondly, it introduces an uncanny valley like effect, drawing attention to the fact that the bot is not real, which would be easier to avoid with just the messaging conversation. However, I will take some parts from it. The ability for the user to interact with, and enter messages, using their voice is a good feature, both to add more ways to interact with the bot, but also to cater for people that may use more accessibility features.



Available Gifts

You need more Koins! Click the Koins button above to purchase them, or earn them by chatting with Kuki.



Figure 7

Features of my solution

My solution will be a chatbot that you can communicate with through a text message like interface. It will use a conversational AI instead of a rule-based approach, as this generates more natural feeling messages. It will use a sequence-to-sequence model with a separate encoder (to analyse the users input and find the semantic information about the user's input) and decoder (which takes in the semantic information from the user's input, and returns the next word in the sequence, in order to generate the response).

It will use a messaging like interface, with a bar to enter a message, and text bubbles to show previous messages. Whilst I could create a new style of interface, users will already be familiar with the interface style used in text messaging platforms, and so it will make it easier for them to understand the UI of my application.

It will also have a speech-to-text option, which will allow users to enter a response by talking into their microphone and use automated speech recognition to convert that into a text input.

The interface would have the option for the user to react to messages with emoji-style reaction bubbles, as well as report messages that they believe to be incorrect (to allow for better debugging and improvement)

The program will also have a search feature, allowing users to search through their old messages, to relive previous conversations with the bot, and see the context in which those messages were sent in.

Finally, users will have the option to report messages, adding a reason onto the report. This allows for easier bug reporting and checking, making future improvements to the program easier.

Limitations

Part of the limitations will stem from the conversational AI model. If this cannot generate very natural responses, the conversation will feel unnatural and less human-like. The model needs to be able to fully understand the user's messages, and be able to form a natural, human-like response as otherwise the user may feel as though the conversation is very inorganic, and may not feel comfortable talking to the bot.

Also, using complex machine learning models, especially pre-trained ones, will take up large amounts of disk space on the user's computer. As this program is fully local (all files will be downloaded on the user's computer) and will not interact with any kind of central server, it will need to store all of the machine learning model used on the user's hard drive. Whilst I could in theory run this on a server, the cost aspect of renting and running a server means this is not possible and would also mean that the program would require internet access for the user, whereas my solution does not require any internet access after the program is downloaded, lowering the barrier to entry for my program to allow more users to use it.

Finally, either the quality of the bot's responses, or the time it takes to generate them, will be heavily dependent on the user's computer hardware. Slower computers, without GPUs with CUDA cores, or with CPUs with slower clock speeds, may struggle to run the program effectively, and so some users may be unable to use the program. However, decreasing the processing power requirements would mean that the program would not work as required, so that is not possible.

Developer requirements

Hardware

- **A fast enough computer.** This must be able to train machine learning algorithms, in a reasonable time, which requires a reasonably fast processor. The standard peripherals of a mouse, keyboard and monitor are also needed for using the software.
- **Microphone.** As the program will have a speech-to-text feature, A microphone is needed, although the standard microphone built into most computers and headphones is good enough.

Software

- **Windows, Mac, or Linux operating system.** These are supported by both python and PyTorch (the python library used for creating and training machine learning models).
- **Python interpreter.** This is the programming language my chatbot will be developed in, as it is good for machine learning, and support many useful pieces of machine learning software, such as PyTorch.
- **Pip.** This is python's package manager/installer, which will be used to install all the packages required to develop the chatbot. By default, this usually comes installed with python.
- **PyTorch.** This is a library for python that is the most widely used for developing and training machine learning algorithms, with support for many different types of algorithms. This is important for creating and training the machine learning model used for the chatbot. Whilst TensorFlow, the other popular machine learning library, is an option, I am going to use PyTorch as it is more pythonic. This means that common debugging tools, even simply print statements, work with PyTorch, whereas TensorFlow requires you to use its own debugging tool alongside normal python ones.
- **Transformers.** This library has pre-trained models for many different typed of transformers, and will be used to access the model, and fine-tune it.
- **NumPy.** This is a python library for doing complex mathematical operations and is also a pre-requisite for all of the other libraries used.
- **Flask.** A web application framework for python that can be used to create the web app for the chatbot. It allows the html files for the website to interact with the python code for the chatbot, sending data in between. This will allow users to enter messages into a html form, which will be processed by the python code, the output from which will be displayed on a new html page.
- **Any other possible packages.** This may not be all the necessary packages needed to develop the chatbot, more may be discovered during development, and they will need to be installed.
- **A web browser.** A web browser is needed to open the web app, as it will be run out of html files. This can be any mainstream web browser, such as Google Chrome or Firefox.

Stakeholder requirements – success criteria

Design

- **Intuitive, simple interface.** An easy-to-understand interface should be used to allow the user to quickly understand how to use the program without the need for complex instructions. This is to make sure users can understand how to use my program as easily as possible, therefore lowering the barrier of entry to use my program, as they should already be familiar with the type of interface I am using. Furthermore, it means there is no need for me to create complex instructions on how to use the program, reducing development time and file size, as using the program should be intuitive and self-explanatory thanks to the interface.
 - **Success criteria** – this can be shown by creating an interface that appears aesthetically similar to other text message platforms, with message bubbles (with different colours for the user and AI messages) and a box to enter the messages.
- **Customisable display.** Users should be able to customise their display, to make it suitable for their needs and preferences. Whilst I could just have one static, unchangeable interface, having the option to tweak and change it means that it can appeal to more users, some of whom may not like the original interface, so the option to change it may mean they are more likely to want to keep using it as some of the other options may appeal to them.
 - **Success criteria** – there should be separate light mode and dark mode styles, with an easy option to change between them.

Functionality

- **A working AI model.** The model should be able to produce understandable text, which makes sense in the wider context of the conversation currently going on between the user and the chatbot. This is the most important requirement, as it is the main function of the program, and the model needs to be able to generate understandable messages in order to make the conversation.
 - **Success criteria** – the model should be able to generate messages that, when tested by the stakeholder, are reported as easily understandable, accurate and relevant to the inputted message from the user.
- **Search feature.** There should be a search feature where users can search through previous messages sent by either them or the bot and can view the context surrounding those messages. This is particularly important as it is likely my program will be made up of static web pages, that reload to show new content, and will also likely not be able to scroll through previous messages. Therefore, whilst adding this isn't strictly necessary and the program could just focus on the current conversation, without a search feature the user may not be able to view past messages/conversations.
 - **Success criteria** – there should be a search box where users can search through their/the chatbot's previous messages and view the conversation surrounding those messages.
- **Reporting.** Users should be able to report messages the bot has sent that make little sense or are inappropriate for the context of the conversation. Without this feature, there may not be an easy way for me to fix any errors the user has, as I wouldn't be able to see the context

of the error – the message the user sent that may have triggered the error, the AI's message to see what kind of error this is and exactly what has gone wrong, or perhaps even the time it took to generate the message, to see if it really is outside of an acceptable time frame. I could also allow users to submit a reason for reporting, so they could describe exactly what they think the problem is, as feedback from users during production is important.

- **Success criteria** – there should be a report button next to the AI's messages, which allows users to report, with a reason, the message. Reporting the message should save it to a separate file, along with the user's message that triggered it, the report reason, and the time it took to generate the AI's response.
- **Speech to text.** Users should be able to enter messages into the model using their voice, with the program recording them through their microphone and converting the audio to text. This feature allows more ease-of-use, as users do not have to be directly sitting at their computer to talk to the chatbot. It is also better for those who find it hard to use a keyboard and would prefer to simply talk to the bot instead. Therefore, whilst I could only let the user enter text through typing into a text box, allowing the user to enter text through their microphone makes the program more accessible.
 - **Success criteria** – there should be an option next to the text-based message input allowing users to use their microphone to submit messages to the model. The program should record their audio with the microphone and run it through a speech-to-text model, to convert it into a format that the AI can use to generate messages. The AI should then generate a message in exactly the same as it would with a text input.

Hardware

- **A decent computer.** A computer with decent specifications will be needed to run the chatbot, as the models will take some computing power to use. The chatbot technically can be run on lower powered computers but will be incredibly slow. The standard peripherals of a mouse, keyboard and monitor are also needed for using the software.
- **Microphone.** As the program will have a speech-to-text feature, A microphone is needed, although the standard microphone built into most computers and headphones is good enough.

Software

- **Windows, Mac, or Linux operating system.** These are supported by both python and PyTorch (the python library used for creating and training machine learning models).
- **Python interpreter.** This is the programming language my chatbot will be developed in. It will need to be installed on the user's computer to allow them to run python programs.
- **All packages used in the program.** All the packages used in the program will be listed in a requirements text file, allowing them to be easily downloaded using pip (pythons package manager and installer).

- **Pip.** This is python's package manager/installer, which will be used to install all the packages listed in the requirements text file. By default, this usually comes installed with python.
- **A web browser.** A web browser is needed to open the web app, as it will be run out of html files. This can be any mainstream web browser, such as Google Chrome or Firefox.

Design

Design diagram

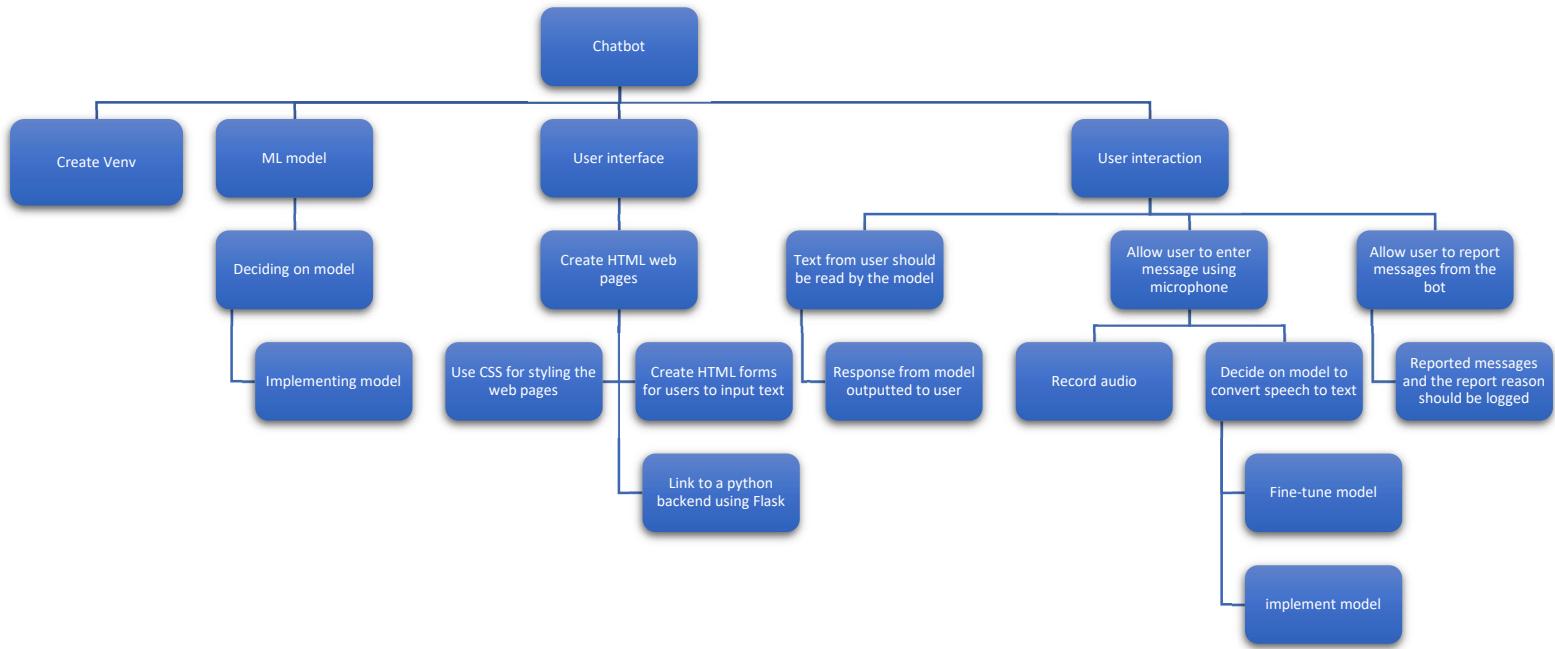


Figure 8

Create Venv

A virtual environment is a folder structure that allows the developer to use a lightweight, but isolated environment. This means that the current python version, plus all modules and libraries used in the development of the program will be stored with the program, independently of all other python modules installed globally.

I could just store all packages globally, however there are a number of issues with this approach, including dependency conflicts. When programming for this chatbot, I would be using a certain version of each of the packages, whilst the packages may be updated in this time – updates which could have conflicts with the versions I am currently using, potentially breaking the program if used. For example, if I were to install these updated libraries for use in another project and hadn't used a virtual environment for the chatbot, these updated libraries (which would overwrite the old ones) could cause conflicts with the chatbot, which had been programmed for earlier versions of the package.

This is not only an issue that would affect me, but also other users who install the program, as these users might have other versions (older or newer) of the packages installed. Users will need to be able to reproduce the environment on their own computers, which will need to not conflict with any possible packages they already have installed.

There is also the possible case that users may be using a computer that does not have administrator privileges, and so won't be able to install packages into the python directory. Therefore, these users should also make use of a Venv.

ML model

Choosing model

Why transformers?

An important part of creating a good chatbot is to choose the right machine learning model. One of the best performing types of machine learning models for natural language processing tasks is the Transformer, introduced in 2017.

Transformers are a type of neural network architecture that has seen wide use in the field of natural language processing, performing better much better than other types of machine learning models like recurrent neural networks. RNN's loop through the language data inputted, finding relationships between data points that it can connect together to gain a better understanding of the input data.

Whilst I could create my own model using TensorFlow and Scikit-learn, Studies have shown that Transformers perform much better across a wide range of natural language tasks. Furthermore, models available from Hugging Face are pre-trained by the big companies and researchers that created them, a process that requires incredibly expensive hardware and trains them to a much better level than I ever could. Therefore, using transformer models from Hugging Face allows me to use models that are a lot more refined and accurate than a more basic model that may make use of a neural network. However, this becomes much more computationally expensive as the input sequences get longer, or more sequences are used, as the RNN tries to compress the sequence into a fixed length vector.

This is where the principle of attention is useful. This means that the model only pays attention to the parts of the input with the most information, therefore allowing the model to maximise the information gained from the input, whilst also being able to handle longer inputs.

Transformers, which use the principle of attention, have shown to have much higher accuracy scores whilst having lower training costs than non-attention-based models across a number of language tasks, therefore showing it to be the best type of model for natural language processing tasks.

What transformer?

Whilst there are a number of different Transformer models, many of them are created and pre-trained for set purposes. For example, Facebook's BART model is best used for summarisation, and Google's T5 is best used for translation.

I have decided to use Meta's BlenderBot, which is created specifically for Conversational AI, the task I need the model for. All iterations of BlenderBot have been shown to outperform other competing models – like Microsoft's DialoGPT – in human evaluation tests. And the latest iteration, BlenderBot 3, performs even better than its predecessors. It is factually incorrect 47% less of the time more up to date (on topical questions) 82% of the time. However, as this is a developing field, it is still not perfect – with 1.2% of users reporting responses as incorrect or nonsensical, and 0.12% reporting messages as off-topic. Overall, whilst it is not perfect it is still by far the best transformer model for conversational intelligence. As it is created and trained specifically for conversational AI, it will produce better results in that specific task that I need it for, as opposed to using a more general-purpose model like Google's BERT. Using a standard, pre-trained version of the model is a lot more practical for my purposes, as Meta states that the model requires a 32GB V100 graphics card to fine tune, which makes it impractical for me.

Implementing model

Transformer models are publicly available through the Hugging Face library, which allows you to access the models and their tokenizers.

The BlenderBot model has different sizes, with the larger size giving higher quality outputs, but being more computationally expensive.

Tokenizer pseudocode

Tokenizers are required to convert the input data, in this case a string, into a type of data understandable by the model, in this case tensors. Hugging Face provides access to the BlenderBot tokenizer, which has been pre-trained by Meta. Meta provides a number of versions of BlenderBot, including a small 90M model, a medium sized 400M model and larger 1B, 3B and 9B models. I am going to use the 400M model as it strikes a balance between good performance and not being too slow. Whilst I could use a larger model, this would make the program run even slower, which would make the programme a lot harder to use for users with lower powered computers. So therefore, as I want to maximise the number of possible users that can use my program, I will use a smaller model.

The name of the model needs to be specified, and the tokenizer should be called, and used to encode the input message.

```
1 name = 'BlenderBot-400M'  
2 tokenizer = BlenderBotTokenizer(name)  
3  
4 input_ids = tokenizer.encode(message)
```

Model pseudocode

The model needs to be called and then is used to generate the ids for the reply tokens.

```
1 model = BlenderBotModel(name)  
2  
3 reply_ids = model.generate(input_ids)
```

Decoding pseudocode

The reply tokens then need to be decoded into a string of text that is readable for humans.

```
1 reply = tokenizer.decode(reply_ids)
```

Overall pseudocode

This can be combined into one function which can be called whenever a message needs to be generated. As this is done as a reusable component, it means less code needs to be written, allowing for quicker development time and simpler looking code.

```
1 Function generate_reply(message):  
2     name = 'BlenderBot-400M'  
3     tokenizer = BlenderBotTokenizer(name)  
4     input_ids = tokenizer.encode(message)  
5     model = BlenderBotModel(name)  
6     reply_ids = model.generate(input_ids)  
7     Return tokenizer.decode(reply_ids)
```

User interface

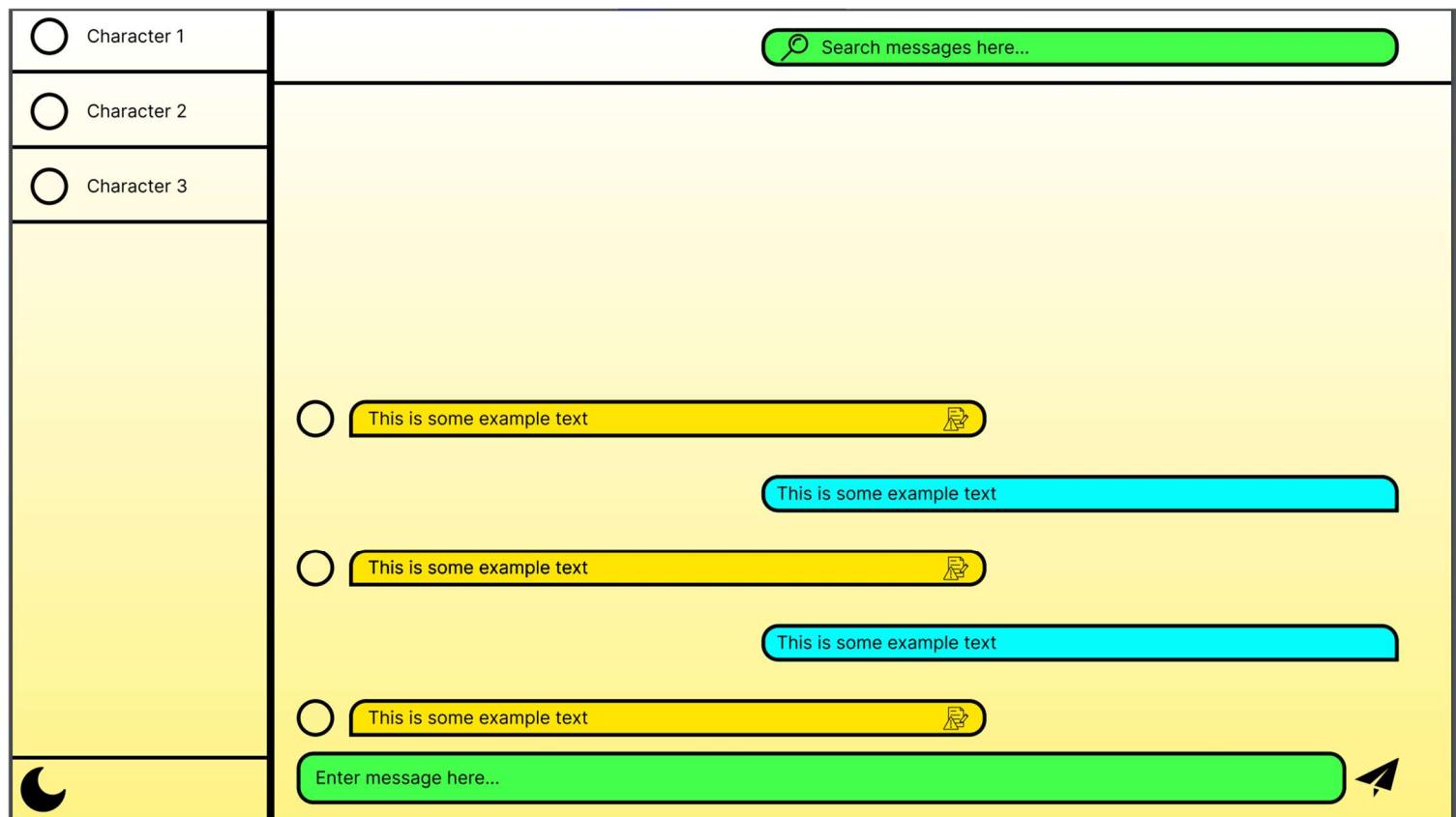


Figure 9

For my user interface, I am going to use multiple web pages, linked to a python backend using the Flask package. Figure 9 shows a mock-up what the light mode version of the GUI will look like.

For my design, I took inspiration from popular chat apps like WhatsApp and Discord. This means that users will already be familiar with the user interface, therefore reducing the time it would take for them to get comfortable using the app. I have kept the interface somewhat basic, in order to make it less cluttered and easier to look at. Apps many elements in their UI, and many things happening on the screen at once, can make the user feel overwhelmed, and too distracted to use the app for its original function.

Some of the blank space left is also useful if I were to add more features to the app in later updates. For example, there is a lot of space in the character list for extra characters, so if I decided to update the app to add more characters the user can talk to, I can just add them on the end. If I had left very little space in the UI, then if I come to add more features, I may have to drastically rearrange the UI, making this choice not very practical.

I have used bright high contrast colours to easily differentiate the different elements in the UI, making everything clear and standing out, as well as creating a design that is more aesthetically pleasing than a more basic design. Whilst I could have just used black text and black borders on a plain white background, not only would this not have looked as good, but the more colourful design is also better for accessibility, as it makes everything stand out from the background, and from other parts of the UI. This makes it easier to see and understand for those with difficulties with sight. I tried to avoid using certain colours, like using both red and green together, to prevent issues for colour blind users. Whilst this does only cater for those with the most common types of colour

blindness, I think this is the more practical as it caters for most colour-blind people whilst still allowing me to make a colourful design.

Create HTML web pages.

The web pages will be written in HTML, with separate divs for each part of the page: the header, the sidebar, the settings panel, and the main section of the page. Each of these divs can then be containers themselves, containing separate divs for their elements. For example, the main div will be a container, containing separate divs for the message input form and the message bubbles.

Use CSS styling for the web pages.

CSS will be used to style the web pages: adding colours and other appearance tweaks, as well as creating grids.



Figure 10

CSS grids are a useful way to arrange and order elements on a web page and can be used to make the design more consistent. I would make the page into a grid, can then create each element to fill a certain number of cells in the grid. An alternative to using CSS grids is to use a flexbox, however



flexboxes only work for one dimension, either in rows or columns. As you can see in Figure 11, the flexbox is only ordering items in rows, with the rows wrapping around to the next line if needed. This is a lot less organised than the grid shown in

Figure 11

Figure 10, which has neatly ordered items by row and column, organising the elements in a much more structured way.

I will use external CSS, keeping all the CSS in one separate file as I will have a constant style across all pages. If I use internal CSS, I will have to make edits to the CSS in every HTML file, often making the same edits each time, whilst external CSS means I just have to make the edits once. It also prevents repetition, as due to keeping a relatively constant styling, using internal CSS would mean I would be adding the same CSS code to every HTML page. This not only means it is harder to test and refactor, but also increases development time and disk space used (although the reductions in disk space would be minimal due to the small size of CSS files in comparison to other parts of the program).

Create HTML forms for users to input text.

There will be two elements where the user can enter text. The first one is the message box, where users can enter text that the model will read and output a response to. The second one is the search

box, where users can search through previous messages. Both of these require the HTML page to take the user input and send it to the python code that can take it as an input, and the HTML page will update to show the output.

[Link to python backend using Flask.](#)

Flask is a python package, which functions as a framework to connect html.

Flask organises the main Python script into separate functions, one for each HTML page, that it runs when the page is loaded. It can use the GET and POST methods to take the input from the HTML forms and store it as a variable in the function. This allows it to be run through the rest of the algorithm. The function can then return the next web page to be loaded and transfer variables from the function to the new HTML page. Flask then allows you to use the Jinja template engine (built into Flask) to output the variables onto the web page. Jinja also allows you to use basic programming tools like for loops in the HTML templates, allowing you, for example, to create a div for every variable in a list.

A popular alternative to Flask is the web framework Django, however Flask has a number of advantages over Django. Firstly, Flask is more lightweight, requiring a smaller codebase, speeding up development time and reducing disk usage. Flask also works better with external packages as it functions as a light, modular framework that creates the web framework and leaves everything else open to the developer. Flask also allows much easier access to the request object – used for access data from HTML forms, whereas Django requires the Request object to be explicitly passed around.

Base Flask pseudocode

```
1 app = Flask
2
3 # home page
4 @app.route('/home')
5 Function index():
6     render('index.html')
7
8 # message page
9 @app.route('/message')
10 Function message():
11     message = request('message')
12     reply = generate_reply(message)
13     render('message.html')
```

Here I have specified functions to be run when the user access the home page and the message page. The home page function will just display the home page for the web app. The message page function will take the inputted message from the html page, use a separate function (to be written later) to generate the reply and then renders the message page, sending the

generated reply onto the page.

Inputting messages via text

Text from user should be read by the model.

Flask's request class provides a number of ways to request data for the flask app, for example 'request.json' parses JSON data, 'request.args' parses data from the URL query and 'request.files' parses in uploaded files. As my html files will contain input forms for the user to enter their message into, 'request.form' is the best way to do this. It requires you to specify the name of the form you are getting the data from, allowing multiple forms to be used. When using this, the type of requests used need to be put into the methods parameter in '@app.route'. 'GET' is the default method and is used to request the html web page with the form. When the user submits data into the form, this sends a 'POST' request, which the function handles.

The data is requested (using the 'POST' request) from the messages HTML form and stored as a variable to be used later in the program.

Form request pseudocode

```
@app.route('/message', methods=['GET', 'POST'])
Function message():
    message = request.form['message']
```

Response from model outputted to user.

The page should be updated to show the message the user sent, as well as the message the model outputted in message bubbles, moving the others up.

Flask uses the 'render_template' method to render HTML pages stored in the templates folder, which is usually used as the return of a function for the original page. The 'render_template' method supports inputting other variables alongside. These variables will be used in the template as Jinja variables, that can be used in the HTML code of the page.

Returning html page pseudocode

```
1 Function message():
2     message = request.form['message']
3     reply = generate_reply(message)
4     Return render_template('message.html', reply)
```

Inputting messages via audio

Users should have the option to submit a message to the user using a microphone. This should be done by having the python program record input from their microphone in the background whilst the user is looking at the html page, and then converting this input into a text string.

Record Audio

When the user selects the microphone option, the python script should record audio and save it as a temporary file in the program folder. The recorded audio will have a set length of 5 seconds. Whilst in theory I could have the user choose to end the recording themselves, it would be much simpler to code for the recording to automatically stop after a set time, and it is unlikely any audio messages would be longer than this (I will test to make sure this is true).

I will use the Sound Device and Sound File libraries to record and save the audio. First, I declare the duration and sample rate of the recording (the sample rate will be 16kHz to match the training used for the Wav2Vec model which will transcribe the recordings). Then I use Sound Device's record function to record the audio, specifying the sample rate and the use of two audio channels. Finally, I will use Sound File to save the audio file, using Sound File's write function.

```
1 sample_rate = 16000
2 duration = 5
3
4 recording = sounddevice.rec((sample_rate * duration), samplerate=sample_rate, channels=2)
5 sounddevice.wait()
6 soundfile.write(file_name, recording, sample_rate)
```

Decide on model to convert text to speech.

To convert the audio file to a readable string that can be used to generate a reply to, another transformer will be used. In this case, Meta's Wav2Vec transformer, a state-of-the-art model for automatic speech recognition. The model uses self-supervised training, being trained on unlabelled data to allow it to achieve the best speech representation possible. It then uses supervised fine-tuning, using labelled data to get better at predicting certain words or phonemes. These approaches, particularly the self-supervised training, give it a massive advantage over other suitable models. Allowing the model to learn good speech representation allows it to achieve accurate results on a small amount of labelled data.

Whilst I could use a different model, if that model does not use the same self-supervised training, it would require a much larger amount of training data to reach similar accuracy levels. Whereas

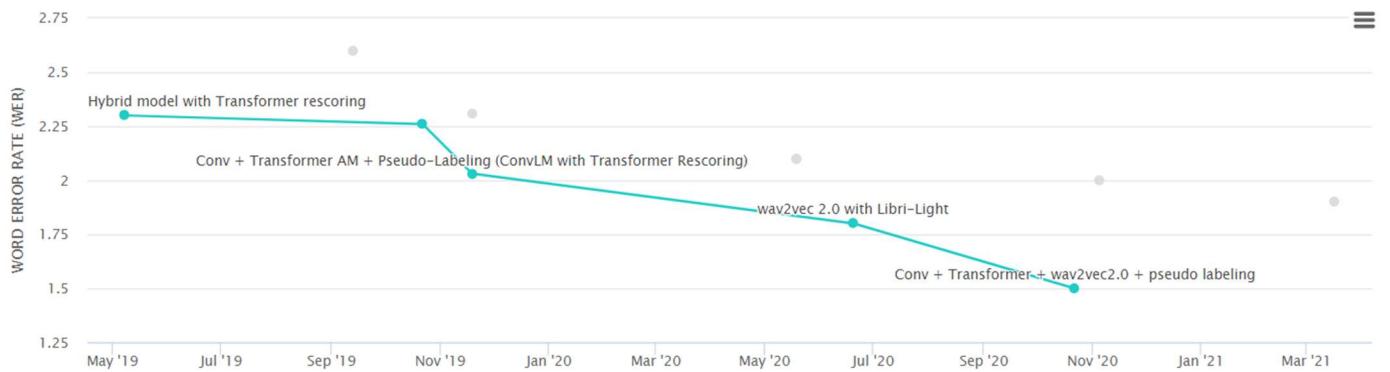


Figure 12

Wav2Vec reaches very good accuracy rates with a dataset 10 times smaller than what is usually used. As you can see from the graph in Figure 12, the latest version of Wav2Vec (2.0) outperforms every competing transformer model, making it the best model for me to use.

Fine-tune model

Fine-tuning a model is the process of training the model on a dataset specific to the task it will be used for, making it perform better at that task.

Whilst I could just use the default, pre-trained version of Wav2Vec 2.0 that is available on Hugging Face, it has been shown that fine-tuning a pre-trained Wave2Vec 2.0 model on even a very little amount of data yields a very small word error rate.

Dataset pseudocode

Fine-tuning a model requires a dataset to train the model on, and Hugging face provides suitable datasets for this purpose. I will use the TIMIT dataset, a set of recordings of 630 speakers, speaking in 8 different English dialects, all reading the same 10 sentences. This makes the dataset particularly useful for me as training it on a wide variety of English dialects means it should be more likely to understand a wide variety of English speakers. It has a version specifically for ASR (automatic speech recognition – the task the model will be used for) that I will use, that is especially good for this task as all the audio files are in the same 16kHz format, allowing me to use a more appropriate, more customised training algorithm.

As well as a standard train-test split of the dataset (which the dataset comes with already), I will also need to split the chunks of speech in the dataset into letters and use this to create a dictionary of the letters used and how many times they occur – using this as the vocabulary the transformer will use.

```
Function extract_chars(data):
    text = ' '.join(data)
    chars_dict = {}
    for char in text:
        chars_dict[char] += 1
    Return chars_dict
```

Before this is done, the dataset will also need to be reformatted to get rid of the special characters found in the text, like punctuation. This is done as these special characters don't have a specific sound linked to them, so the model would struggle to link them to the audio.

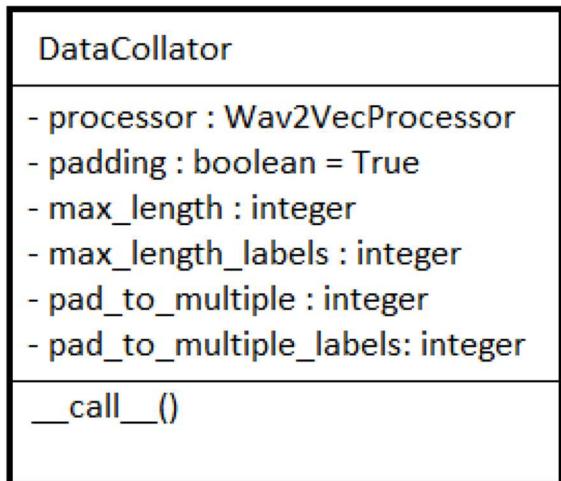
```
1 Function remove_characters(data):
2     data = re.sub(['!', '.', '?', '!', ';', ':'], data).lower() + ' '
3     Return data
```

Creating data collator pseudocode

The first step to creating the training algorithm is to create a data collator. Data collators are used to combine many samples from a dataset into one batch, which can then be processed by the processor I created earlier and used to train the model. The data collator used to fine-tune Wav2Vec is different from other data collators in that it will need to dynamically pad the samples, ensuring they are padded to match the longest sample in their batch, instead of the overall longest sample. This is particularly necessary due to the differences in length of Wav2Vec's input samples in comparison to its output samples.

To do this, I need to apply separate padding functions to the labels and the input values, as input speech and output speech are of different modalities.

The following class will be created for the data collator:



This diagram shows the data collator class will have a number of attributes. Firstly, padding will specify to the collator whether or not the returned sequences should be padded. Max_length sets the maximum length of the input values of the returned data, and max_length_labels sets the maximum length of the labels of the returned data. Finally, pad_to_multiple will pad the sequences to a multiple of the given value.

Figure 13

The call method will be defined as:

```
1  function __call__(self, features):
2      input_features = []
3      label_features = []
4      loop for feature in features
5          input_feature = {"input_values": feature['input_values']}
6          label_feature = {"input_ids": feature['labels']}
7          input_features.append(input_feature)
8          label_features.append(label_feature)
9
10     batch = self.processor.pad(
11         input_features,
12         padding=self.padding,
13         max_length=self.max_length,
14         pad_to_multiple_of=pad_to_multiple,
15         return_tensors='pt'
16     )
17
18     with self.processor.as_target_processor():
19         labels_batch = self.processor.pad(
20             label_features,
21             padding=self.padding,
22             max_length=self.max_length_labels,
23             pad_to_multiple_of=self.pad_to_multiple_labels,
24             return_tensors='pt'
25         )
26
27     return batch
```

Firstly, the inputs and labels are split as, as stated above, they have to be of different length and will therefore be padded differently. I then use the processor's pad method to pad first the inputs and then the labels, using their respective maximum length and padding arguments. As I am using PyTorch, I will also specify that the padding process should return PyTorch tensors.

Evaluation metrics pseudocode

During training, the model should be evaluated using the word error rate. Loading the word error rate will return a sequence of logit vectors, which will contain the log-odds for each word in the vocabulary dictionary I created earlier. As I want the most likely prediction, I will take the argmax of the logits. I will also decode the prediction ids and label ids.

```
1  function compute_metrics(prediction):
2      prediction_logits = prediction.predictions
3      prediction_ids = np.argmax(prediction_logits, axis=-1)
4
5      prediction_string = processor.decode(prediction_ids)
6      label_string = processor.decode(prediction.label_ids)
7
8      wer = wer_metric.compute(predictions=prediction_string, references=label_string)
9
10     metric = {
11         "wer": wer
12     }
13
14     return metric
```

Loading model

```
1  model = Wav2VecForCTC.from_pretrained(
2      'facebook/wav2vec2-base',
3      ctc_loss_reduction='mean',
4      pad_token_id=processor.tokenizer.pad_token_id
5  )
```

Here I load the pre-trained Wav2Vec model from Hugging Face.

Training arguments pseudocode

Here I define all the parameters related to training the model.

Group by length groups samples of similar lengths together into one batch, speeding up training by reducing the number of padding tokens that are passed through the model. Per device train batch size specifies the size of the batch that will be run through each CPU core. Setting the evaluation strategy to 'steps' means that the evaluation is done at a set rate (set by the eval_steps) parameter. Gradient checkpointing is used to save memory, which is useful due to the lower computing power I have access to. However, this does mean the backward pass will be slower. The use of fp16 mixed precision training reduces the also saves memory, with the trade-off of having a lower precision. This is a trade-off I feel is necessary to ensure the algorithm can run smoothly on the lower computing power.

Any arguments not explicitly specified will use their default value, and the values I have used are simply estimates, and will be

```
1  training_arguments = TrainingArguments(
2      output_dir=output_dir,
3      group_by_length=True,
4      evaluation_strategy='steps',
5      per_device_train_batch_size=16,
6      num_train_epochs=20,
7      gradient_checkpointing=True
8  )
```

tweaked based on trial and error when developing the training algorithm.

Creating trainer pseudocode

Now all of these aspects can be passed into a trainer, which can then be used to train the model.

```
1 -> trainer = Trainer(  
2     model=model,  
3     data_collator=data_collator,  
4     args=training_args,  
5     compute_metrics=compute_metrics,  
6     train_dataset=timit_processed['train'],  
7     eval_dataset=timit_processed['test'],  
8     tokenizer=processor.feature_extractor  
9 )
```

This trainer can then be run using `trainer.train()` to run the training algorithm on the model.

[Implement model.](#)

Now that the model has been trained, it should be used to transcribe the audio message that has been stored as a .wav file.

Firstly, the audio file should be loaded, and for this I will use the Sound File library that I used earlier to save the recorded audio files.

Then I will use the fine-tuned processor to process the speech, getting the input values the model requires to transcribe the audio. Then the input values will be run through the model, extracting the logits of the predicted values. As I want the most likely prediction, I take the argmax of the logits. Finally, I use the processor to decode the predicted logits, returning a transcription that should be in readable English and match the audio fed into the model.

```
1 -> Function asr_transcript(input_file):  
2     speech = soundfile.read(input_file)  
3  
4     input_values = processor(speech, return_tensors='pt').input_values  
5     logits = model(input_values).logits  
6     predicted_ids = torch.argmax(logits)  
7     transcription = processor.decode(predicted_ids)  
8  
9     return transcription
```

Allow users to report messages.

There will be an option next to every message from the bot to report the message. When clicked, a popup will appear with a HTML form in. There will be a text box for the user to input their reason for reporting the message.

The popup will be created using a Bootstrap modal. This requires a trigger – a button that the user must click to open the modal. The modal part contains a header, main content, and a footer (which will contain a button to close the modal). The main content will contain the HTML input form, which will have a textbox, allowing the user to enter a report reason, and a submit button. The reason for using a Bootstrap modal is that there needs to be a form with a text box in it allowing users to enter a report reason, and this text box does not fit on the main page, therefore requiring some form of pop-up box. Whilst I could create my own, using Bootstrap's modal will speed up development time, whilst also being well-suited for my use-case.

Reported messages and the report reason should be logged.

The program will then take in the reported message, and find the previous message sent from the user. The reported message, the user's previous message and the report reason will be stored in a CSV file.

This allows for easier error reporting, as I would simply need the CSV file to see how the model made an error and could use that information to change the model's training if needed.

Testing

Throughout the development of the program, each function and feature should be tested when it is created, to ensure that it works as intended. Therefore, this means that once the program is completed, it should all fully work, as each part would have been tested independently. The other benefit of iterative testing throughout development is that it makes it easier to fix bugs and errors in individual parts of the program whilst they are still being developed on before any changes may conflict with other parts of the program that use this function.

Testing should include stating the data input into the program, the intended output – what the program should be outputting (usually by rendering a page in the web app) – and the actual output of the program. If there are any errors, those should be shown from the console.

After development is carried out, A final full test of the program should be carried out, to ensure that there are no extra errors or bugs, that either may have been missed in the iterative testing or have perhaps appeared due to conflicts when combining different functions that were tested separately.

Finally, the program will be sent off to stakeholders, who will test not only for bugs and errors, but also if it matches their expectations for features and usability. I will send them a copy of the program, exactly how any other user would receive it when in production, to ensure that I can get feedback on the full user experience, including setting up the program, along with a questionnaire asking them to provide feedback on specific parts of the program, such as the reporting or search features, as well as usability aspects like the installing process. This will be an important part as it will see if the completed program matches the stakeholder's requirements.

Iterative tests

Message response generation via text

After each update to the algorithms pertaining to the use of the machine learning model to generate responses to user's messages, I will iteratively test it. To do this, a selection of sentences will be inputted into the message generation function, initially by hardcoding it into the program by calling the function in its python file, and then once the web app and its HTML forms are developed and connected to the python algorithms, I will use that to input sentences. The selection of sentences should vary in type, with some being questions asking for an opinion, whilst others being statements that the model should react to and be able to further the conversation using. I will then add the kind of response I think the model should generate, before showing the model's actual response. I will then discuss the model's response, providing feedback on it based on how understandable and human-like the response is, how well it fits the input sentence, and how well it furthers the conversation.

Web app development

After each update to a page on the web app, or an additional page being added, the web app should be opened and used to get the edited page to be rendered. This means that whenever an edit is made and needs to be tested, the web app should be opened from the home page, and I should navigate through the pages, exactly how a user would, until I find the web page that needs to be tested, and test that.

When testing, as the input, I should record exactly what inputs I am making to the web page, be that entering a string of text into a text form or clicking a button. I should then record exactly what changes I am expecting to the web page, or exactly what the next web page that will be rendered as a result of my inputs should look like. Finally, I should show the changes to the web page, using screenshots, as the actual output.

Message response generation via audio

The testing process will be similar to the process for message generation via text, and as both algorithms will overlap, with the message generation via audio converting the audio to text before using the other response generation algorithm that will already have been tested to generate a response to the audio input. Therefore, as it has already been tested, the actual response generation algorithm doesn't need to be as rigorously tested. However, once the audio recording and saving has been tested, the algorithm for converting it to text should be tested in conjunction with the response generation algorithm. Whilst I could test them independently, testing the audio-to-text convertor, and then testing the message generation algorithm using the converted audio-to-text as input, this would be unnecessary, and could perhaps lead me to seeing the algorithm as worse than it is. This is because I do not need the speech-to-text algorithm to be perfect, I just need it to be able to create a string of text where the original intended meaning can be accurately understood by the response generation algorithm.

When testing, as input I should write exactly what I will speak into the microphone, which would be a variety of different types of sentences for each test. I will then enter what the algorithm should output, which may be a correct audio recording, or a correctly generated message using a relatively accurate transcription. I will then record the actual output from the algorithm, either by copying a print statement from the console, showing a screenshot of the web app, or confirming what was in an audio recording. In the case of testing message generation, I will also record the audio-to-text transcription. Whilst this isn't necessary in the case that the program works, as what I am testing is the response being generated, in the case that the response generated is wrong, it would be useful to see what the transcription was, and therefore what was inputted into the response generation algorithm, as it would help work out what algorithm created the error.

Development

Development diagram

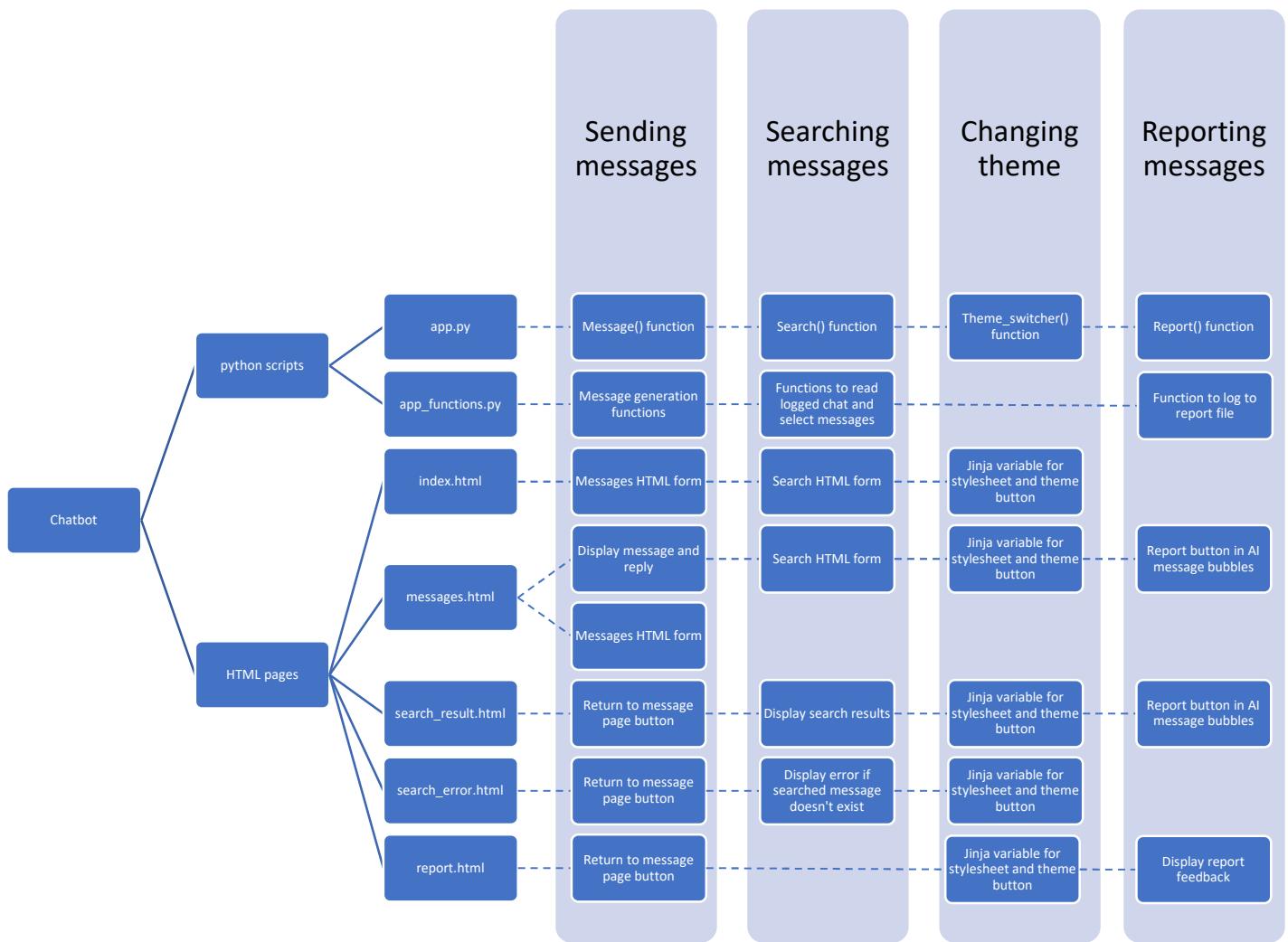


Figure 14

Development practices

Python version

This project is explicitly being written in Python version 3.9.9. Whilst when I started development there were more recent versions of Python available (Python 3.10), neither PyTorch nor TensorFlow support for that version yet, and wouldn't function properly. More updated versions of python may work with it, and I can check once development has finished whether the program will still work on updated versions, but the program is written primarily for Python 3.9.9.

Git

When developing my project, I am taking advantage of a service called git, a distributed version control system. It stores a history of all changes made to the program in a repository. This is useful as it allows me to go back and revert to a previous version of any of the files in the event of any unchangeable mistakes or corrupted files.

It also works as a backup, as any of these versions are downloadable from a separate server, so if a device used for development gets lost, stolen or broken, I can still access the code for the project by downloading the most recent version from the git repository. Whilst there are other backup options available, such as OneDrive, Google Drive or simply doing a physical backup using an external storage media, these don't come with the version control feature – saving a new version of some of the files in the project would overwrite the previous versions, therefore not allowing me to go back and fix any mistakes that might be in the most recent version, or seeing other changes I made previously. Also, as git is a very common part of developer's workflow, it has good support among most IDEs, meaning that it is simple and easy commit changes to a git repository, without having to open another file.

GitHub

To easily use git and access my repositories, I use a service called GitHub – an online hosting platform for git repositories. It allows me to view my repositories, as well as a commit history for each file, as shown in Figure 15.

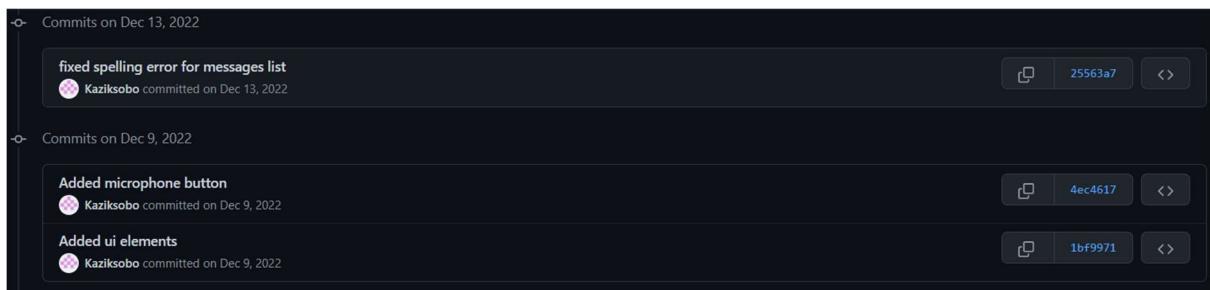


Figure 15

It also gives the ability to see exactly what edits each commit made to the file, highlighting added code in green and removed code in red. This allows me to quickly identify what changes may have caused a bug I am experiencing, as well as allow me to quickly revert small parts of the code without having to restore the entire file to a previous commit. An example of this is shown in Figure 16.

```

17 17      </div>
18 18      <div id="header">
19 19          <form action="/search" method="post" autocomplete="off" id="search-form">
20 20 -      <input type="text" id="search-bar" placeholder="Search messages here..." list="messages-list">
20 20 +      <input type="text" id="search-bar" placeholder="Search messages here..." list="messages-list">
21 21          <datalist id="messages-list">
22 22              {% for message in messages_list %}
23 23                  <option value="{{ message }}">

```

Figure 16

Type hints and Docstrings

Throughout the code, I use function annotations and docstrings to add clarity to my code and the functions I am writing.

Function annotations work as type hints, showing the data type a variable should be. I use annotations purely for function arguments, as this is where they are the most useful. Whilst I could use annotations for all variables, I don't find there is much use in using them for mundane things like variable declarations, as it is obvious what data type the variable will be from what the variable is being defined as. On the other hand, with function arguments, it is often not obvious what the variable is meant to mean, and therefore what should be inputted when calling the function.

Another reason why these are useful is that they help prevent errors. In a lot of the functions I will write, inputting an argument with the wrong data type would lead to python throwing up an error, as I would be trying to do an operation designed for a different data type. This gets even worse when it is not an obvious error. For example, a function to add two numbers together would not throw an error in python if both the arguments (that in theory should be the numbers to be added) were strings, as adding two strings together is a valid operation in python (string concatenation). Therefore, the output would be wrong, but I would not find out about the bug until the function's output was used in a different operation. Using annotations prevents this, as it makes it obvious exactly what type of variable this function is designed for, therefore giving more clarity on potential inputs and the use of the function.

However, these are only type hints are exactly that, only hints. As python is a dynamically typed language, simply using these types on their own doesn't enforce type checking. Python will still let you use a different type for an argument, even when there is an annotation explicitly stating what type to use. Whilst there are libraries such as Mypy that can use type hints to enforce explicit type checking at compile-time, I haven't made use of those. I feel the type hints are enough to tell me what to be used and using more libraries in development has downsides: more libraries could potentially mean more libraries for users to download, even if the type checking wouldn't be of a massive use to them. Adding libraries means that the download process for my program might be more complex, and will definitely be longer, with the overall program (including any used libraries) being larger, increasing download time and the storage space the program will take up on users' computers.

I also make use of docstrings. Doc strings are strings literals at the start of the function (straight after the definition) that can be used to create a description of the function, or to give advice on how it is to be used. Not only can this be used to give a deeper explanation, but it could, for example, be used to show an example of the type of output that should be expected from this function.

Below, you can see an example of annotations and docstrings in use, with a function I have written while developing the program. As you can see, there is an annotation to show that the argument

‘message’ is expected to be a string, and that the output of the program is expected to be a flask.response object. The docstring provides a brief description of the use of the function, not being particularly long due to the simple nature of the function.

```
def message(reply: str) -> flask.Response:  
    """Displays the main page updated with the latest message"""  
    return render_template('message.html', message=reply)
```

I don’t have to use docstrings or annotations, as comments could be used instead. However, comments usually serve a different use – they are mostly meant for people who read your code with the intent to extend it, whereas docstrings are usually used for people who intend to use your code, without needing to know how it works. As the project is not intended for other contributors, users would not need to know how exactly any specific function works, simply how they should be used.

Furthermore, annotations and docstrings are easy to access for functions without having to find the function’s definition. In python, all functions have a `__doc__` method, which shows any possible docstring, and a `__annotations__` method which shows any annotations. These mean that the annotations and docstrings can be accessed from other points in the file, other files or even the command line. Therefore, if I used comments, to see the information in the comment explaining how to use the function, I would have to search for the function definition itself, which is particularly tedious on larger projects like this which span multiple files. On the other hand, annotations and docstrings can be accessed from anywhere, therefore reducing development time, as less time needs to be spent searching for previously written explanations.

In fact, my chosen IDE (Visual Studio Code), and most others, shows the annotations and docstring contained in the function whenever any instance of the function is hovered over, as seen in the example below. This would not work if comments were used instead, therefore being another way where the use of annotations and docstrings can help clarity and speed up development time. An example of the annotations and docstrings being displayed for the previously shown ‘message’ function can be seen below.

```
(function) message(reply: str) -> Response  
    Displays the main page updated with the latest message  
    message()
```

Flask algorithm

The main algorithm will be written with in python, using the flask library. Flask allows me to combine the html template pages together and use python algorithms and functionality to display data on the html pages. I use Flask's requests class to request data from specific input boxes, referring to them with their id. I can then render a chosen HTML page, using a number of python variables which will turn them into Jinja variables.

Jinja turns HTML pages into templates, where Jinja code can be used to create basic algorithms. This means that the python code can be used to change the output of a html file. If this wasn't a feature, every time new data was inputted into a form, and processed by a python code which would return an output, a new html file would be needed. This minimises the HTML files that need to be written, saving development time and file space, and also allows the program to be more easily expanded. The drawback is the Jinja is not a full programming language, and as such can only write basic pseudocode-style algorithms. This means that almost all actual programming needs to be done in the python scripts, with the Jinja code only deciding what HTML to output.

Flask also requires a set file structure, where the HTML and CSS files are kept in separate templates and static folders respectively. The static folder also contains any other assets required for the web pages, such as images.

Base structure

```
1 import flask
2 from flask import Flask, render_template, request
3
4 app = Flask(__name__)
5
6 @app.route('/')
7 def main() -> flask.Response:
8     """Displays the home page 'index.html'"""
9     return render_template('index.html')
10
11 if __name__ == '__main__':
12     app.run(debug=True, port=5000, host='0.0.0.0')
```

Here I have used basic, boilerplate Flask code to create a working webapp that simply displays the main page of the app.

First, I import the Flask library, allowing me to use all parts of the library, and then explicitly imported specific methods. This means that I have access to all of the library, whilst still having a shorthand for certain regularly used methods.

The third line creates an instance of a flask app, which will serve as the webapp that the program will run as.

I then create a decorator on line 5, which wraps the function below. @app.route connects the endpoints of URLs to the code contained in the function it is wrapping. In this case, when the URL with the route ('/') is searched for, the main() function is called. As '/' is the default endpoint of all

URLs (even if you enter a URL without an endpoint, it will simply default to '/'), the main function is used to display the landing page – the first page users will see when running the app.

Lines 11 and 12 tell Flask to start the development server when the program is run. This is not intended for users to use, but instead provides a quick and easy way for the developer to see the changes new code has made to the webapp.

I have also added basic structure for the message and search functionalities to be added later on:

```
11  @app.route('/message')
12  def message() -> flask.Response:
13      pass
14
15  @app.route('/search')
16  def search() -> flask.Response:
17      pass
```

As with the message function, both of these functions use a decorator to specify what URL endpoint will call them. These URL endpoints are triggered by the HTML form actions, so when a user submits an input into the HTML form with the action '/message', it goes to that URL endpoint, therefore calling the message() function.

TEST – message input

As shown by the diagram in **Error! Reference source not found.**, this is testing a feature which also uses the messages.html and index.html files. The code written in those files, which is used for this feature, is explained below.

As shown in the code below, I have added some basic functionality to the message function, that will be run when the user submits an input into the message input form.

```
11  @app.route('/message')
12  def message() -> flask.Response:
13      user_message = request.form['message-input']
14      print(user_message)
15      return render_template('message.html')
```

This uses Flask's request object to request the input from the form with the name 'message-input'. It will then print it to the console, it will then render the message.html page. As the functionality to display messages on that page hasn't been written yet, this should just display the default page. However, this still allows me to test the rendering functionality, and is necessary anyway as Flask requires these functions to have a return. This allows me to test three features at once: the ability of the HTML input form to take in user input, the ability of the Flask algorithm to request that input and the ability of the Flask algorithm to render a second HTML page.

Input – entering 'This is a test message' into the message box in the web app.

Expected output – the python script printing out ‘This is a test message’ into the console, and the message.html page loading in the browser window.

Actual output –

Method Not Allowed

The method is not allowed for the requested URL.

Figure 17

As shown in Figure 17, I have received a 405 error – method not allowed, when I entered text into the message box and clicked enter.

With research, I have found that this was because I did not put the URL methods in the @app.route decorator. A GET request is a common HTML method that I would need to use to retrieve information from the Flask web server, and a POST request is often used to send data to the web server. When a user submits input, the POST method will send the data to the Flask web server, and the GET method will send the information to the python script. As you can see here in the HTML form, I have included the POST method, specifying that as how the data will be sent.

```
32 |         <form action="/message" method="post" autocomplete="off" id="message-form">
```

Changes to make – include the GET and POST methods in the @app.route decorator, as shown here

```
11 |     @app.route('/message', methods=['GET', 'POST'])
12 |     def message() -> flask.Response:
13 |         user_message = request.form['message-input']
14 |         print(user_message)
15 |         return render_template('message.html')
```

This should allow the methods needed for the requested URL, therefore not showing the error, and instead printing the message input.

2nd iteration

After making the changes, ‘This is a test message’ was outputted to the console, showing that the algorithm to request and process data from input forms works.

```
This is a test message
```

HTML pages

Home page – CSS grid

1st iteration

Here I have created the CSS grid that shows the layout of the home page. This fits the structure of the mock-up I created in the design section.

Styles.css:

```
6 .grid-container {
7   display: grid;
8   grid-template-columns: 15vw 70vw 15vw;
9   grid-template-rows: 9vh 81vh 10vh;
10  gap: 0px;
11  background-color: #2196F3;
12  padding: 0px;
13  width: 100vw;
14  height: 100vh;
15 }
16 .sidebar {
17   grid-column: 1;
18   padding: 5px;
19   grid-row: 1 / 3;
20 }
21 .header {
22   grid-column: 2 / 4;
23   padding: 5px;
24   grid-row: 1;
25 }
26 .main {
27   grid-column: 2 / 4;
28   padding: 5px;
29   grid-row: 2 / 4;
30 }
31 .settings {
32   grid-column: 1;
33   padding: 5px;
34   grid-row: 3;
35 }
36 .grid-container > div {
37   background-color: rgba(255, 255, 255, 0.8);
38   text-align: center;
39   padding: 20px 0;
40   margin: 5px;
41   font-size: 30px;
42 }
```

index.html:

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="utf-8">
5     <link rel="stylesheet" href="static/styles.css">
6     <title></title>
7   </head>
8   <body>
9     <div class="grid-container">
10       <div class="sidebar">sidebar</div>
11       <div class="header">header</div>
12       <div class="main">main</div>
13       <div class="settings">settings</div>
14     </div>
15   </body>
16 </html>
```

I have created a div called grid-container, which contains all the items that will be contained in the grid. I have set the exact size of each row and column inside the grid-container element, using responsive units (vw and vh). This means that the left and right columns will take up 15% of the page, and therefore the middle column take up 70% of the page. The rows work in the same way, except with the top row taking up 9% of the page, the middle 81% and the bottom 10%. I then specify how many columns and rows each grid item should take up. For example, the sidebar panel is in column 1, and rows 1 to 3 and the settings panel is in column 1 and row 3. As I have set the width and height of the grid container to 100vw and 100vh respectively, the grid container should take up the entire page.

Whilst I could have used a different measurement for sizes, such as pixels, instead of vw and vh, the page would not have been responsive. Whilst I could fine-tune the pixel values so that it works on my computer screen, with a 1920x1080 resolution, anyone viewing it with a different resolution monitor, or anyone that may have simply resized their browser window, would see a jumbled up, poorly arranged page. Therefore, I have used vw and vh to ensure the elements in the page scale in accordance with the page's dimensions.

Instead of using padding, I was originally going to make use of the 'gap' property of CSS grids. 'gap' allows me to set a universal spacing between grid elements giving a more visual separation between parts of the page. Whilst I want this effect, I found that using the 'gap' property didn't work, as it made the grid would be bigger than the page, meaning the user would have to scroll down to see the whole page. This happened because the grid element's sizes added up to 100% of the page, so adding gaps between them made the grid size more than 100% of the page. I fixed this problem by using padding. Each of the grid elements has 5px of padding, meaning that there is still a visible separator between the grid elements, but the separator is inside the elements, and so doesn't add to the size of the page.

Output:

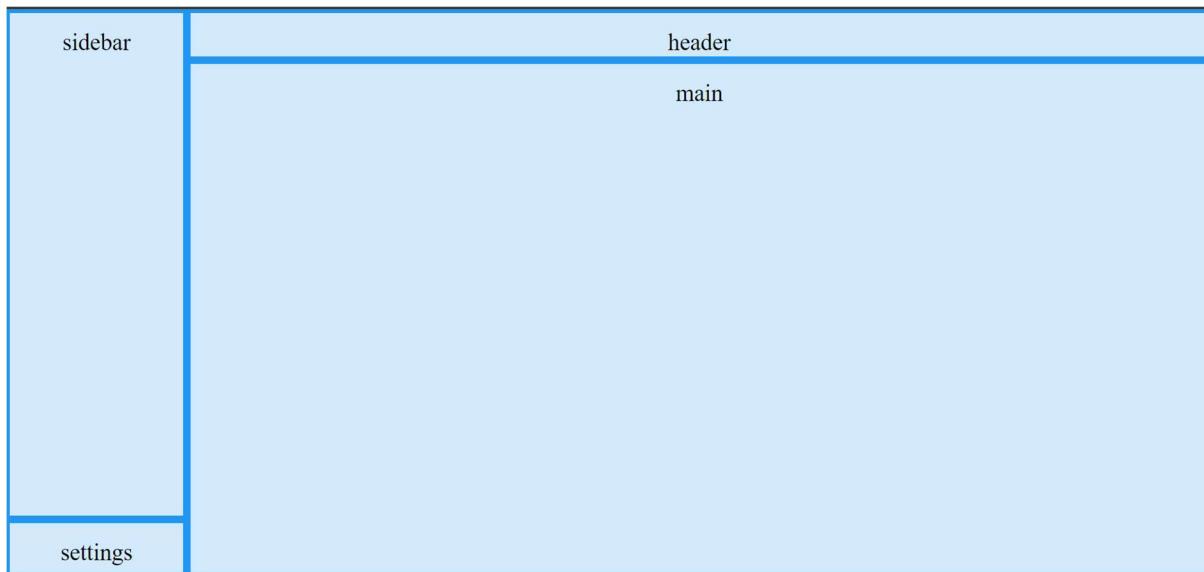


Figure 18

2nd iteration

```
1 body{  
2     overflow: hidden;  
3     margin: 0;  
4     padding: 0;  
5     background-image: linear-gradient(#FFFFFF, #FFF384);  
6 }  
7 .grid-container {  
8     display: grid;  
9     grid-template-columns: 15vw 70vw 15vw;  
10    grid-template-rows: 9vh 81vh 10vh;  
11    gap: 0px;  
12    padding: 0px;  
13    width: 100vw;  
14    height: 100vh;  
15 }  
16 .grid-container > div {  
17     text-align: center;  
18     font-size: 30px;  
19 }
```

```
20 .sidebar {  
21     grid-column: 1;  
22     grid-row: 1 / 3;  
23     border-left: 0vw;  
24     border-top: 0vw;  
25     border-right: 0.25vw solid #000000;  
26     border-bottom: 0.25vw solid #000000;  
27 }  
28 .header {  
29     grid-column: 2 / 4;  
30     grid-row: 1;  
31     border-top: 0vw;  
32     border-right: 0vw;  
33     border-bottom: 0.25vw solid #000000;  
34     border-left: 0.25vw solid #000000;  
35 }  
36 .main {  
37     grid-column: 2 / 4;  
38     grid-row: 2 / 4;  
39     border-right: 0vw;  
40     border-bottom: 0vw;  
41     border-top: 0.25vw solid #000000;  
42     border-left: 0.25vw solid #000000;  
43 }  
44 .settings {  
45     grid-column: 1;  
46     grid-row: 3;  
47     border-bottom: 0vw;  
48     border-left: 0vw;  
49     border-top: 0.25vw solid #000000;  
50     border-right: 0.25vw solid #000000;  
51 }
```

Firstly, I have now added a background to the webpage, making use of the 'background-image' property to add a linear gradient the same as that in the mock-up.

I have also changed from using padding as a separator to using a border. Whilst I could have stayed with padding, as it fit the purpose it was needed for before, as shown above, using a border brings a number of advantages. Firstly, the border property gives you more specific control, allowing me to set borders only on certain sides of the page elements. This has allowed me to only give a border to the inside edges of elements, leaving the edges of the page free of borders. It is also easier to give borders a colour independent of the rest of the page, unlike with padding.

Output:

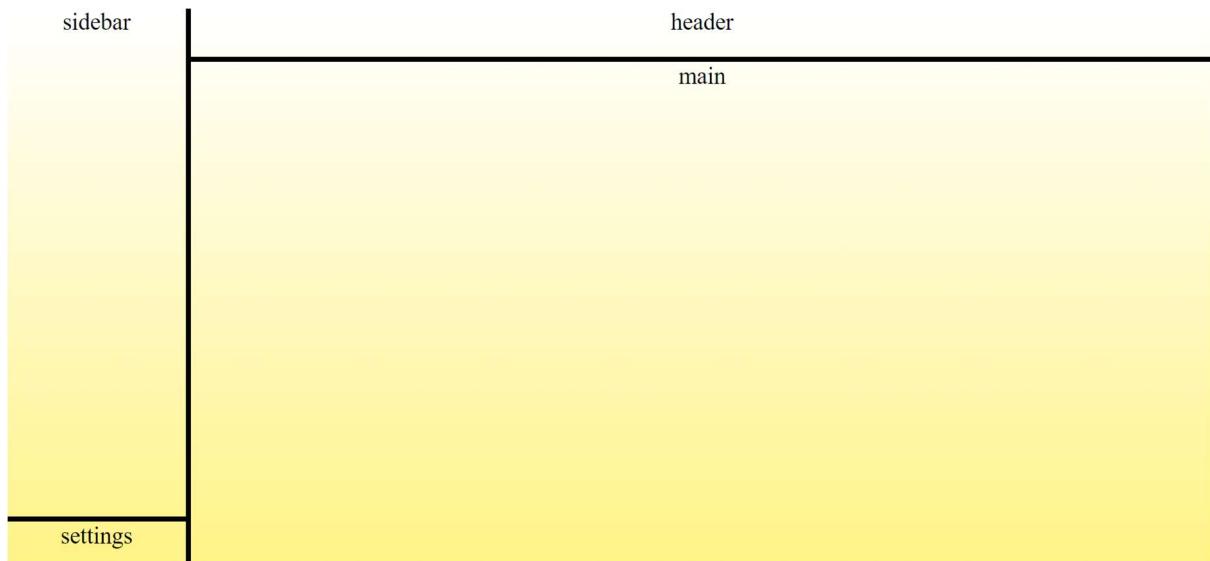


Figure 19

Home page – adding elements



Figure 20

I have added a message input bar along the bottom using the following HTML code:

```
<div id="main">
  <form action="/message" method="post" autocomplete="off" id="message-form">
    <button type="submit" id="microphone-button">
      
    </button>
    <input type="text" id="message-input" placeholder="Enter message here...">
    <button type="submit" id="message-submit-button">
      
    </button>
  </form>
</div>
```

Here I create a HTML form with the '/message' action, so it will send the input to the /message page, which is read by the Flask code in app.py and is expected to display a page with the /message extension. The input form will send the input using the post method, and the form consists of three elements – a text box, a submit button and a microphone button. The submit button contains an image, and CSS is used to make the button only take up the image. I have also used alternative text for the image and will do this with every image used. This helps with accessibility, as it means that for those with screen readers, their screen readers will be able to describe the image, therefore meaning they won't miss out on any information they may need from the image. The microphone is also a submit button, but this time it will not send any data to the flask app instead just triggering one of the functions to display a page. The button contains an image, and again CSS has been used to ensure the button appears to only be the image. Again, for accessibility reasons, the image makes use of alternative text.

CSS:

Firstly, I added the 'display: flex' property to the main div, turning it into a flex container, with the message input form as a flexbox inside. I use the following CSS to style the message form:

```
#message-form {  
    border: none;  
    flex-direction: row;  
    margin-top: auto;  
    text-align: center;  
    display: grid;  
    grid-auto-flow: column;  
    grid-template-columns: 4vw 72vw 4vw;  
    grid-template-rows: 6vh;  
    margin-bottom: 3vh;  
    gap: 1vw;  
}
```

As I have specified the flex direction as row, the form flex box will be a row inside the main div. As margin-top is set to auto, the rows will be arranged from bottom up, therefore meaning that the form will be a row on the bottom of the page. I have then set the message-form div to be a grid container. I have used relative units (vw and vh) to create a grid with 3 columns and 1 row, each with set (relative) sizes. I use the 'grid-auto-flow: column' property to ensure that the grid only takes up

one row. This works because this property makes items flow across a single column, instead of going down in rows. I specify 'text-align: center' to ensure that the message input and submit buttons are in the centre of their grid boxes. I also used the gap property to add a small gap (again using relative units) between the grid boxes.

The following CSS is for the text input and submit button:

```
#message-input {  
    background-color: #44FF4C;  
    border-radius: 12px;  
    border-color: #000000;  
    border-width: 3.5px;  
    padding: 10px;  
    padding-left: 12px;  
    font-size: 2.5vh;  
}  
  
#message-submit-button {  
    background: transparent;  
    border: none;  
}  
  
#message-submit-icon {  
    height: 100%;  
    width: auto;  
}
```

The text box is given a border, with the radius creating curved corners, giving a smoother look. It is given a separate background colour, to allow it to be set out from the rest of the page, and easily visible, with good contrast. This allows those with accessibility concerns to still be able to easily separate the message box, and other elements on the page, from each other and the rest of the page. I have used relative units to set the size of the font used, meaning that the font will shrink when the page shrinks, therefore meaning that the element should stay (somewhat) consistently sized in respect to the rest of the page

For the submit button, I have set the background of the button to be transparent, meaning that the only part of the button that can be seen is the message icon. This approach isn't perfect, as the

button itself is still a square, the user could click slightly to the side of the icon and still be pressing the button, however I couldn't find a better way to do it.

I have also set the height of the icon image itself to 100%, so it should completely fit the dimensions of the button. Without doing that, the image would revert to its original dimensions with no relation to the overall size of the button, which would make it look hilariously oversized, taking up a large portion of the page.

The submit button isn't strictly necessary, as the user can submit any text entered into the text box simply by pressing the enter key, however the submit icon gives a more explicit way to submit text, and makes it clear the function of the message bar.

I have used the following CSS to style the microphone button:

```
#microphone-button {  
    background: transparent;  
    border: none;  
    padding: 0;  
    margin: 0;  
}  
  
#microphone-icon {  
    height: 100%;  
    width: auto;  
}
```

Similarly to the submit button, I made the background transparent and removed the border so that it appears as if the button is only the microphone icon, and not in fact a box surrounding it. I have also removed any margins and, this is to allow the icon to be centred, taking up the same vertical space as the message input bar and submit button. This hasn't worked perfectly, as you can see in the image below. When the page is shrunk, the submit button stays centred but the microphone button appears to be a bit too high up. However I couldn't find a way to fix this, and I feel as though users drastically resizing the height of the page is an

rare enough scenario that this shouldn't be significantly noticed by users.



Figure 21

However, this approach wasn't working responsively. When I resized the page, shrinking it, the spacing between the form elements became inconsistent, with the microphone icon having basically no separation with the message bar, whilst the send icon still has a strict separation, therefore giving it no margin to the edge of the page.

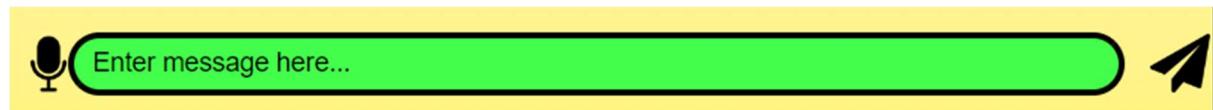


Figure 22

To fix this, I changed how I added the microphone and submit icons. Instead of using the `` tag inside the buttons, I left the content of the buttons empty and added the icons as background images. As you can see from the CSS below, I added the images as background images. I then set the background colour to be transparent, as otherwise a white box the size of the button will appear behind the image. I also ensure that the image is centred inside the button, and strictly contained inside of the dimensions of the button, ensuring it doesn't expand out.

```

228     background-image: url(https://cdn2.iconfinder.com/data/icons/font-awesome/1792/send-512.png) !important;
229     background-color: transparent;
230     background-repeat: no-repeat;
231     background-position: center;
232     background-size: contain;

```

I have also added a message search bar using the following HTML code: .

The form has the '/search' action and the post method, so it will use that method to send the input to the Flask code, which is expected to display a page on the /search extension. The input form contains two elements – a text input and a Datalist. The text input works in the same way to the

```

<div id="header">
  <form action="/search" method="post" autocomplete="off" id="search-form">
    <input type="text" id="search-bar" placeholder="Search messages here..." list="messages=list">
    <datalist id="messages-list">
      {% for message in messages_list %}
        <option value="{{ message }}>
      {% endfor %}
    </datalist>
  </form>
</div>

```

message input above, except that as there is no submit button. Whilst I could have added a submit button for the same reason that I added a submit button for the message bar (added clarity), I felt like that would have added unnecessary clutter to the header, and I added a (not clickable) search icon inside the search bar which should make the function clear enough. I used the following CSS to create the search bar:

```

#search-bar {
  background-color: #44FF4C;
  background-image: url(https://www.iconpacks.net/icons/2/free-search-icon-3076-thumb.png);
  background-position: 10px center;
  background-repeat: no-repeat;
  background-size: contain;
  padding-left: 30px;
  border-radius: 12px;
  border-color: #000000;
  border-width: 2.5px;
  padding: 4px;
  padding-left: 60px;
  height: 3vh;
  width: 40vw;
  margin-right: 4vw;
}
::placeholder {
  color: #000000;
}

```

I added a search icon to the left of the search bar by adding it as a background image. I could then position it 10px from the left edge of the search bar, whilst centring it vertically. I then specify the property 'background-repeat: no-repeat' as otherwise the background image would repeat as many times as it can fit inside the element.

Similarly, to the message input, I use a different colour to the page background, as well as a rounded border, to separate it from the rest of the page, partially for aesthetic reasons and partially for accessibility reasons.

I added a sidebar to the left-hand-side of the page, which is set out so that it could show a list of other chatbots that the user could communicate with. Whilst now, there will only be one, setting the page out like this means that in the future adding selection boxes for other chatbots.

To do this, I used the following HTML code:

```
<div id="sidebar">
  <div id="chatbot-select">
    
    <p id="chatbot-name">
      Chatbot
    </p>
  </div>
</div>
```

This creates a div, which itself contains an image (the chatbot icon) and the chatbot's name (currently just chatbot).

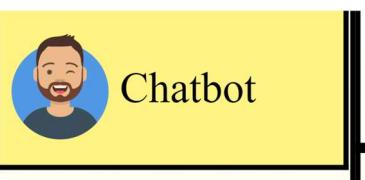
I then styled it with the following CSS code:

```
#chatbot-select {
  border-bottom: 1px solid #000000;
  border-right: 1px solid #000000;
  width: 100%;
  display: grid;
  grid-template-columns: 35% 65%;
  grid-template-rows: 10vh;
  grid-auto-flow: column;
  text-align: center;
  background-color: #FFF384;
}

#chatbot-icon {
  height: auto;
  width: 80%;
  margin-left: 10px;
  margin: auto;
  display: block;
}

#chatbot-name {
  text-align: left;
  margin-bottom: auto;
  margin-top: auto;
  font-size: 2.8vh;
}
```

I added a border to separate that chatbot select section from the rest of the page, as well as any other options that may be added in the future. I set the chatbot-select div to take up the entire width of the sidebar using the property 'width: 100%', as otherwise there is a small gap between the chatbot-select div and the border of the sidebar, as seen in Figure 23. I then set the div to be a grid container, with the chatbot icon and



name being grid boxes inside. The container consists of two columns and one row, with their dimensions relative to the size of the container. I have also given it a different background colour, allowing it to stand out from the rest of the page and showing that it is the chatbot that has been 'selected'.

Figure 23

I set the icon to take up 80% of the width of its grid box, which I felt was a good size that didn't make the chatbot take up too much space height-wise. This is because, as I have set the height of the image to auto, allowing it to scale proportionally to the width to prevent image distortion, using a larger width would have meant the height would have been bigger as well, taking up too much of the page. I use the 'margin: auto' property to centre the icon inside its grid box, however for this to work I also need to include the 'display: block' property, which converts the

image (which is an inline element by default) to a block-level element. Also, a width needs to have been set so that the left and right margins can take the remaining space and auto-align themselves.

For the chatbot's name, I use a relative font size, the same as in the message input box. I set the bottom and top margins to auto to centre it vertically, and then align the text so that it starts from the left edge of its grid box.

I have also added a settings panel, which for now just includes a dark mode icon, using the following HTML code:

```
<form action="/dark" id="dark-mode-form">
  <button type="submit" id="dark-mode-button">
    
  </button>
</form>
```

The settings div contains a form that uses the '/dark' action, however, it doesn't use any method to send data as no data is sent. It simply triggers a function in the flask program which should display a different page, without the need to send any data. The form only contains a single element – a submit button that contains an image – the dark mode icon.

I use the following CSS to style the button:

```
#settings {
  grid-column: 1;
  grid-row: 3;
  border-bottom: 0vw;
  border-left: 0vw;
  border-top: 0.25vw solid □#000000;
  border-right: 0.25vw solid □#000000;
  display: flex;
  justify-content: left;
  align-items: center;
}

#dark-mode-button {
  background-color: transparent;
  border: none;
}

#dark-mode-icon {
  height: 8vh;
  width: auto;
}
```

The settings div becomes a flex container, which uses the 'justify-content: left' property to ensure that the flex boxes go across the container from left to right, and 'align-items: center' to ensure that the items are vertically centred. As there is only one element, it is situated on the left side of the container.

As with the message submit button, I set the background to be transparent and remove the border, so it appears that the button is simply just the image. I then give the image a height relative to the size of the page and set the width to auto to scale the image properly.



Figure 24

Creating the messages page

The messages page will be the same as the index page, except containing the messages inside the main div. It will use a container div, which will contain individual divs for each message.

```
<div id="main">
  <div id="messages-container">
    <div id="message-1" class="ai-message-box">
      <p class="message-text">
        Hello
      </p>
    </div>
    <div id="message-2" class="user-message-box">
    </div>
  </div>
```

Here I have created a div inside the main div called 'messages-container'. Inside it is currently a div for a message from the bot, and a div for a message from the user. These will be user generated in the future, but to test the HTML and CSS, I am currently using hardcoded versions. This currently just creates an empty, bordered box above the message form, as shown in Figure 25.

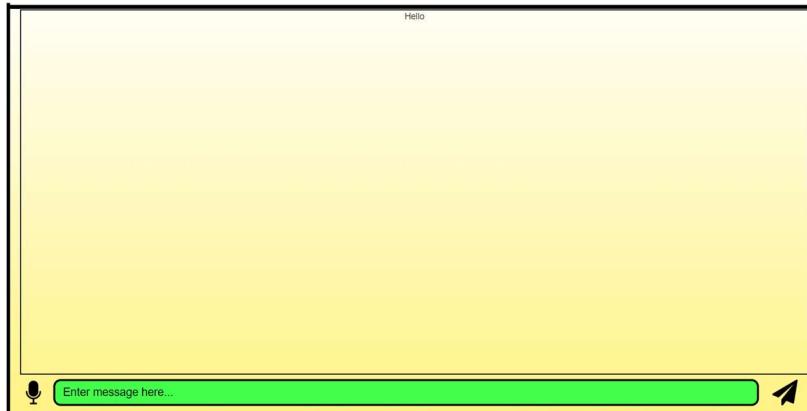


Figure 25

Messages boxes

1st iteration

Each message written by either the user or the AI should be shown on the screen in the form of message bubbles, as shown in the design mock-up in Figure 9. This mock-up shows rows of message bubbles, with the most recent at the bottom and the least recent at the top. There will be a maximum number of text boxes shown, which will be until the screen is filled – 4 user messages and 5 chatbot messages. Whilst it may be a useful feature to allow the user to infinitely scroll up through past messages, I found that this didn't work with method I was using to display the messages. I am using a CSS grid, which doesn't allow for a grid of infinite size, where the size of the list of items is unknown. Furthermore, the need for an infinite scroll through messages is negated by the search feature, which allows users to search through their past messages. If anything, this is a better way of doing it, as users can quickly find a specific message, rather than having to scroll for a while to find the message they are looking for.

For the HTML side, I have written 9 divs like below for the message boxes, 4 with the class user-message-box and 5 with the class ai-message-box.

```
60 | <div id="message-2" class="user-message-box message-box">
61 |   <p class="message-text">
62 |     user message
63 |   </p>
64 | </div>
```

All of these boxes are in a div called messages-container, which will act as a grid container. Using the CSS below, I have turned messages-container into a grid with 1 column and 9 rows, each with a height of 11%. Unlike in a lot of the rest of the CSS used throughout my program, I used percentage

```
36 #messages-container {
37   grid-row: 1;
38   margin-left: 1vw;
39   margin-right: 1vw;
40   display: grid;
41   grid-template-columns: 100%;
42   grid-template-rows: 11% 11% 11% 11% 11% 11% 11% 11% 11%;
43 }
```

units instead of vh and vw units. This is because vh and vw are relative to the entire size of the viewport (the area of the page viewable to the user), whereas percentage units are relative to the size of the div. Using vh would mean that the boxes would extend below the input form, not just staying in the container div. Also, as the size of the container div is already relative to the size of the viewport, the design will still be responsive.

This HTML and CSS produces the page shown below. In Figure 26, you can see that the message boxes are equally divided up in terms of space inside the container div.



Figure 26

2nd iteration

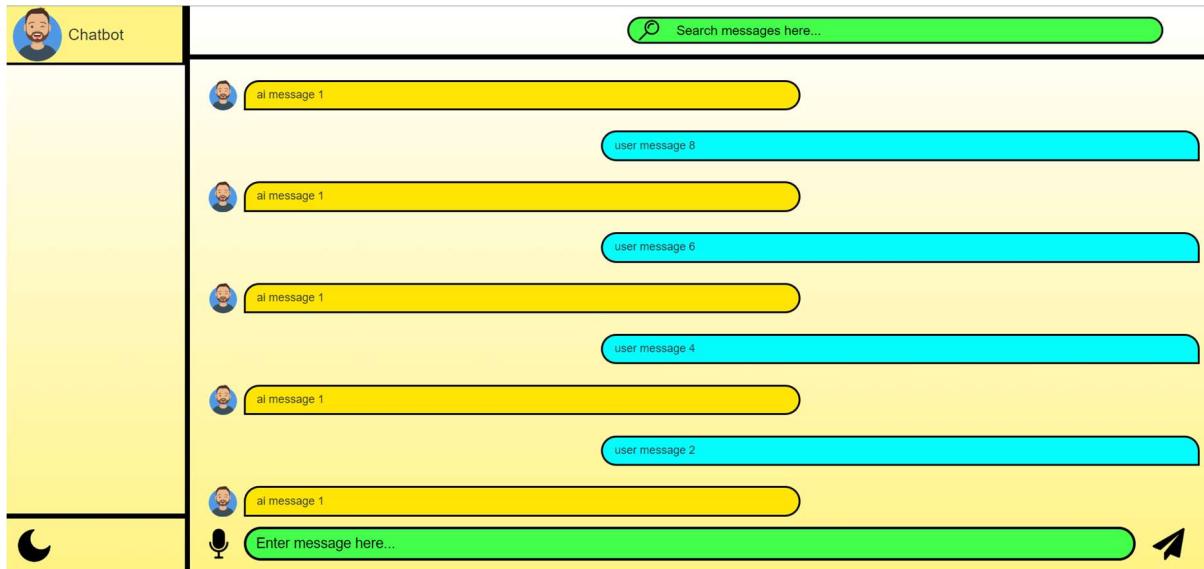


Figure 27

Here I have turned the grid boxes into actual message bubbles. The following is the HTML for one of the AI's message bubbles:

```
<div class="ai-message-container" id="message-1">
  <div class="ai-message-icon">
  </div>
  <div class="ai-message-box message-box">
    <p class="ai-message-text message-text">
      ai message 1
    </p>
  </div>
```

The AI's message bubbles are contained inside a container div, which contains both the message bubbles and the chatbot icon, to show that the message is from the chatbot. To create this, I made the ai-message-container div a grid item, and then gave it a width and height of 60%, to ensure the message bubbles are not too big, and to give appropriate spacing between each one. The CSS below

56	margin-left: 0;	moves the container to the bottom left corner of the grid space, ensuring it is separated from the user's messages (as it is aligned to the other side of the page), and again giving it an appropriate gap to the message above. I then
57	margin-right: auto;	
58	margin-top: auto;	
59	margin-bottom: 0;	
60	display: grid;	turn the container into a grid container with one row and two columns, sized as
61	grid-template-columns: 7% 93%;	7% of the container and 93% respectively.

As before, whilst these are not vh or vw units, they will still be responsive to the overall page size due to the responsive nature of the container. In grid column 1 is the chatbot icon. Whilst originally, I tried to use a div which would contain and tag for that chatbot icon, I couldn't get this version to work. The image kept being ludicrously big, far outsizing the grid square supposedly available for it. Instead, I simply created a div for the image, and set the background of that div to the chatbot icon, as follows:

```
72 .ai-message-icon {
73   grid-column: 1;
74   background-image: url(https://www.pngall.com/wp-content/uploads/12/Avatar-Profile-Vector.png);
75   background-repeat: no-repeat;
76   background-position: center;
77   background-size: contain;
78   padding: 0;
79   margin: 0;
```

Using line 77, I specify that the image should be contained inside the div, not overflowing outside of it, and centre it using line 76.

The following is the HTML for one of the user message bubbles:

```
64 |         <div id="message-2" class="user-message-box message-box">
65 |             <p class="user-message-text message-text">
66 |                 user message 2
67 |             </p>
68 |         </div>
```

The user's message boxes are bare grid items, with being inside a container, as there is only one element to the box – the message bubble itself, rather than a secondary element like the chatbot's icon, that would have to be put alongside it using a grid. As with the AI message box, I set the height and width to 60% for the same reason and used the below CSS to align it to the bottom right corner

```
83 |     margin-left: auto;
84 |     margin-right: 0;
85 |     margin-top: auto;
86 |     margin-bottom: 0;
```

of the grid space, further spacing it from the AI message boxes. Whilst this extra degree of separation isn't strictly necessary, it is another way to visually set the user's and AI's message boxes apart. It is particularly useful for users with smaller screens, as, due to the responsive design practices prevalent through the design of the web page, shrinking the window still has the different boxes physically separated from each other. Another way I separated the different message bubbles is by using different colours. Using vibrant, visually distinctive colours easily allows users to separate the two types of message bubbles from each other and is particularly useful for users with poorer vision.

I use the below CSS to position the message text in the vertical centre of the boxes, accompanied by

```
92 |     position: absolute;
93 |     top: 50%;
94 |     -ms-transform: translateY(-50%);
95 |     transform: translateY(-50%);
```

the 'position: relative' attribute, used by all message bubbles. Explicitly using the '-ms-transform' property ensures that the styling will work the same on all browsers a user may access the web page from, including internet explorer. Ensuring compatibility for all browsers like this lowers the barrier to entry for using my web app, therefore ensuring that the greatest number of users possible will be able to use it and all see a consistent styling.

```
101 | #message-1 {
102 |     grid-row: 9;
```

Finally, I use the following CSS for all message boxes, whereas the number of the box increases, the grid row decreases. This means that the boxes will be ordered from bottom up.

Bootstrap modal

I now want to add in the report icon for the AI messages. This will be done using a bootstrap modal, which is a popup window displayed on top of the page. Bootstrap is a third-party framework that includes template for HTML and CSS design. There are other options to create pop-up boxes, like jQuery dialogs, but I found that using Bootstrap modals is a much simpler way of doing it and requires much less code to be written as you can just make use of the pre-written CSS Bootstrap provides.

Html:

```
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
```

```
<button type="button" class="btn btn-info btn-lg report-button" data-toggle="modal" data-target="#report-modal">
</button>
<div class="modal fade" role="dialog" id="report-modal" tabindex="-1" aria-labelledby="report-modal-label" aria-hidden="true">
  <div class="modal-dialog vertical-align-helper">
    <div class="modal-content">
      <div class="modal-header">
        <h4 class="modal-title">
          Report message
        </h4>
        <button type="button" class="btn btn-default modal-close-button" data-dismiss="modal"></button>
      </div>
      <div class="modal-body">
        <form action="/report" method="post" autocomplete="off" class="report-form">
          <input type="text" class="report-reason-input" placeholder="Enter report reason here...">
        </form>
      </div>
    </div>
  </div>
</div>
```

This modal is not created using my own CSS and is instead created using Bootstrap. Therefore, to allow me to make use of Bootstrap elements, I added the Bootstrap stylesheet, adding it above my own stylesheet, as shown below. Adding it above my own stylesheet means that my stylesheet has priority, therefore meaning any CSS I add to style Bootstrap elements will overwright Bootstrap's own styling. Bootstrap's CSS is used by referencing Bootstrap's classes. An example of this is shown in line 42 when the Bootstrap class 'modal fade' is referenced.

```
5
6
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.4.1/css/bootstrap.min.css">
<link rel="stylesheet" href="static/styles.css">
```

First, a button to activate the modal is created, with the class referencing a class from the bootstrap style sheet, which is required for this to function.

Then the report modal is created. The div with the modal fade class uses bootstrap CSS to add a fade effect to the rest of the page whilst the modal has popped up. Whilst this isn't necessary for the modal's functionality, and I could just not include it, it adds a bit of extra emphasis to show the user exactly what part of the page they should focus on and interact with. It also adds a visual hint as to where you can click to close the modal.

The modal content div includes inside it all the content that will be contained inside the pop-up box: a heading and an input form. The header contains a title for the modal – 'Report message', and a close button. The close button is a button that makes use of Bootstrap's 'btn', 'btn-info' and 'btn-lg' classes, as well as the 'data-toggle' and 'data-target' attributes to control the modal. It is an empty button, as it will be given a background image with CSS.

The input form will call the '/report' action, using the POST method, in order to send the input data to the function in the python script with the '/report' decorator. The form simply contains a text input box for users to enter their report reason, and the form will submit when the user presses enter, whilst typing into the text box. Whilst I could add a separate submit button, I don't think this is necessary. The form will submit when the enter key is pressed, whether there is also a submit button or not, making the submit button functionally useless. Also, unlike with the message input form, I don't think it looks stylistically better to have a separate submit button.

However, there was an issue where using the modal was, for some reason, changing the sizing of the search box, making it drastically smaller when I added the bootstrap styling. I found that this was because it was, for some reason, making use of the box-sizing property that it had gotten from bootstrap. To fix this, I added a 'box-sizing: content-box' to my own CSS for the search box, which took precedence over bootstrap's property. Content-box just sets the box sizing to the default, which is different to the border-box setting it was getting from bootstrap. I am still unsure as to why

it was using bootstrap styling for the search box, because, as far as I know, I was not calling any bootstrap templates.

I then added my own CSS to style the modal, first styling the activator button. First, I turned the AI message bubbles into CSS grid containers using the below CSS code. This splits the message bubble

```
73     display: grid;
74     grid-template-columns: 93% 7%;
75     grid-template-rows: 100%;
```

into a grid with two columns – one having a width of 93% of the bubble's width, and the other having 3% of the width.

I then specify that the text in the AI message bubbles should be in column one and use the below CSS to style the report button, including specifying that it should be in column two.

```
108 .report-button {
109     grid-column: 2;
110     background-color: transparent;
111     background-image: url(https://static.thenounproject.com/png/1236835-200.png) !important;
112     border: none !important;
113     background-repeat: no-repeat;
114     background-position: center;
115     background-size: contain;
116     padding: 0;
117     margin: 0;
118 }
```

I set the background of the button to a report icon image, and centred it using line 114, as well as ensuring it only takes up the space inside the button using line 115. I used the '!important' argument when setting the background image and removing the border, to ensure that it overwrites any Bootstrap styling, as I found that without that, there was still a border and a white background as this was part of Bootstrap's default styling modal buttons. Leaving the border there would, however, have uses – the clickable area of the button is a rectangle and not just the area of the PNG background image, so users could click slightly off the image, and it would still work as a button. Therefore, the border has a use as it shows users explicitly where to click. However, I feel like the button works out better without the border, as it looks better stylistically and it makes it harder for users to miss-click, by making the button still work even when they didn't click exactly on the image (chances are if they are clicking right next to the image, they intended to click the button).

I then repeated the below CSS code for all different states of the button (active, hover, focus and

```
120     background-color: transparent !important;
121     border: none !important;
122     outline: none !important;
123     box-shadow: none !important;
```

visited) as this was necessary to overwrite all of the Bootstrap styling and ensure a consistent design even when clicked on.

Next, I created styling for the actual modal and its content. Firstly, I vertically centred the modal, meaning that when it pops up after the button is clicked, it appears in the middle of the page. To do this, I wrapped the modal content in a container with the class 'vertical-alignment-helper', and added the 'vertical-align-center' class to the next div, as shown by the HTML code on the page above. I then used the CSS below to make these classes render the modal box in the middle of the page.

I turned the ‘vertical-alignment-helper’ class into a table, with the div inside (‘vertical-align-center’) being a cell in the table. I then use line 151 to vertically centre the div. Then, I inherit the height and

```
143  .vertical-alignment-helper {  
144    display: table;  
145    height: 100%;  
146    width: 100%;  
147    pointer-events: none;  
148  }  
149  .vertical-align-center {  
150    display: table-cell;  
151    vertical-align: middle;  
152    pointer-events: none;  
153  }  
154  .modal-content {  
155    /* Bootstrap sets the size of the modal in the modal-dialog class, we need to inherit it */  
156    width: inherit;  
157    max-width: inherit; /* To avoid the modal window stretching full width */  
158    height: inherit; /* To center horizontally */  
159    margin: 0 auto;  
160    pointer-events: all;  
161    background-image: linear-gradient(■#fffce, ■#fff7af);  
162  }
```

width of the modal-content class from Bootstrap’s ‘modal-dialog’ class. I also inherit the maximum width from the same class to avoid an issue where the modal window stretched to the full width of the page for some reason.

I use CSS to turn the modal header class into a grid container, with one row and two columns – one taking up 90% of the width and the other using the other 10%. The title is in grid column one, and uses a padding (measured in percentage of the overall width of the modal) to place the title relatively in the centre of the modal. The close button is in column 2, and similarly to the activator button, uses a background image, as well as specifying that the background colour would be transparent, the image will not repeat and will be centred and contained in the button, as well as having no border. This gives helps give a smooth, modern look to the modal. As with the activator button, I repeat these arguments for the focus, active and hover states of the button, to ensure a consistent styling at all times. When doing this, I also make use of the ‘!important’ attribute again, ensuring that in the case of any conflicts with Bootstrap’s default styling, my CSS takes priority. This is important as I found Bootstrap likes to add white backgrounds, as well as borders, outlines and box shadows to buttons. Therefore, the ‘!important’ arguments is useful as it ensures the button is always styled the way I intended, not accidentally using any of Bootstrap’s styling.

The report reason input form uses a similar styling to the other text input forms on the web page (the search bar and message input bar), with a green background, black border with rounded corners, and padding to give a comfortable amount of spacing between the text and the edges of the text box. This means that the text box won’t feel too crowded or packed together.

The result of this is AI message bubbles that look like the following, and a modal that is centred in the middle of the page with the appropriate styling, as shown below in Figure 28.



Figure 28

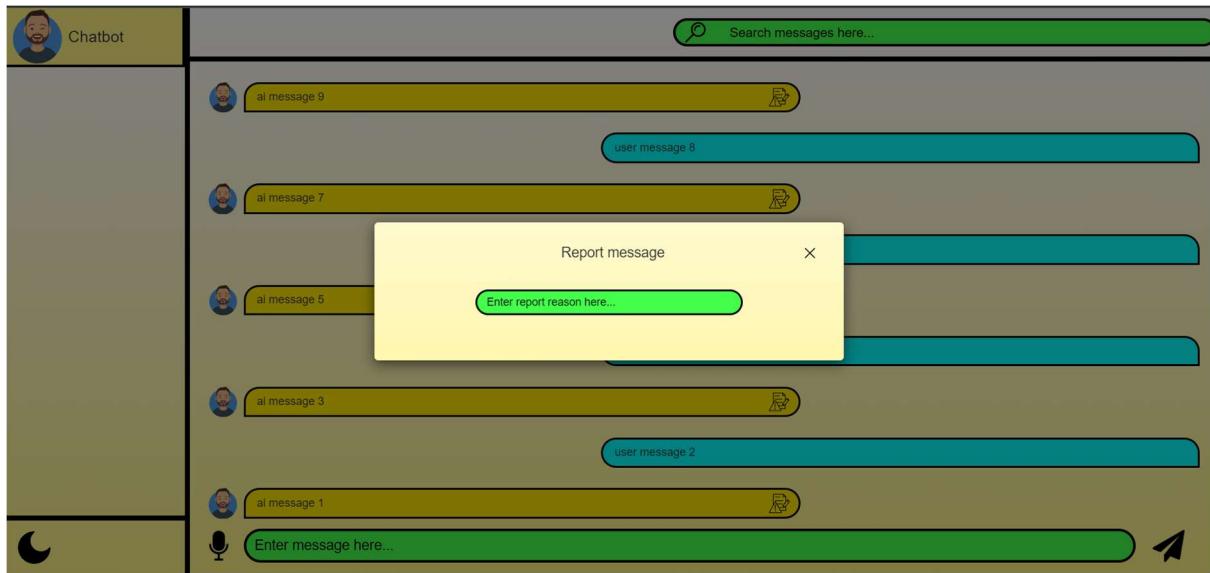


Figure 29

The results of the modal styling is shown in Figure 29, with the modal being centred, both vertically and horizontally, in the middle of the page. The dimmed area outside the modal area is clickable, and will close the modal.

Creating dark mode page

To create the dark mode page, I will use a separate CSS style sheet, which is largely similar to the original style sheet, but with different colours. On the whole, the colours will simply be the opposite of what they currently are, e.g., switching black text with white text.

There are other possible approaches, such as using different HTML files, in which are different class names that would link to other styles on the original CSS style sheet. Or I could have simply turned the class names into Jinja variables, and sent different variables with flask, depending on if the user was in dark mode or light mode.

However, these approaches have downsides. The first approach would involve making more files, and files which are larger in size, in comparison with a second style sheet. This is due to the simple fact that there are more HTML files, and they are larger in size, with the messages.html file currently at 15KB for example, whilst the CSS stylesheet is only at 8KB. Therefore, adding more HTML files would increase the file size for the user more than adding an extra stylesheet. Based off this, turning class names into Jinja variables would seem like a smarter solution. However, this becomes impractical as the number of classes and ids used increases and would also make the python file larger.

Before creating the dark mode stylesheet, I ensured all images were added with CSS properties, instead of in the HTML files using the `` tag. This is because the icons will have to be changed due to them being black. I will change them to inverted versions, with the black outlines to be changed to white outlines. Therefore, as the only thing that changed when the user changes from light to dark mode is the stylesheet, all the images will need to be stored in the CSS stylesheet. For example, the colour mode switch icon shown below was stored in the `` tag inside the button in both the HTML files. This was changed to be a background image in CSS, using the same properties

as I have shown before for adding background images, including setting a transparent background colour. I also set the width and height to be 100%, to ensure that the button still fits inside the settings div (the bottom of the sidebar).

I then created a new CSS file, dark-styles.css, which will contain the stylesheet used for the dark mode version of the page. The stylesheet is the same as the previous one, except the colours are flipped and the images are changed.



Figure 31

Figure 32



Figure 30

I then added the functionality for the user to switch between light and dark mode. To do this, I added a function in the python script which will render the current page with the correct theme, as shown below.

```

21  @app.route('/theme', methods=['GET', 'POST'])
22  def theme_switcher() -> flask.Response:
23      """Switches the theme of the web app"""
24      if current_theme == 'light':
25          current_theme = 'dark'
26          stylesheet = 'static/dark-styles.css'
27      else:
28          current_theme = 'light'
29          stylesheet = 'static/light-styles.css'
30      return render_template(current_page, stylesheet=stylesheet)

```

Separately, I declared variables for the current theme and current page, which will be set to 'light' and 'index.html' by default respectively. The function above checks what the current theme is when the button is pressed, and switches to the other theme (e.g., if the current theme is 'dark', it changes the current theme to 'light'). It also creates a variable to store what stylesheet to use, storing the file path of either the light or dark CSS stylesheets. This means that it is easy to use them when

rendering the page. I

```
<link rel="stylesheet" href="{{stylesheet}}>
```

then render the current page (as the current page will be set to a file name (e.g., 'index.html')), sending the `stylesheet` variable as a Jinja variable called `stylesheet`. Below is how it would be used in the HTML file, with the curly brackets around it signifying that it is a Jinja variable and not referencing anything else or to be interpreted as a string for where to find the `stylesheet`.

The ability to use Jinja for templates is one of the main advantages of Flask, as it allows me to easily edit static HTML files, based on changes in the python script, simply by sending variables across.

As the python function is inside the `app.route` decorator that listens for the '/theme' URL endpoint, the function will only be called once the HTML form with the action '/theme' submits. As, unlike the message input form, no data is actually sent from the HTML form, I do not use Flask's `request` module to request any data from the form, the only use of the form is to trigger the function.

TEST – theme change

As shown by the diagram in **Error! Reference source not found.**, this is testing a feature which also uses the `messages.html` and `index.html` files. The code written in those files, which is used for this feature, is explained below.

Here I will test if the previously explained code works. Initially I will be on the index page and be using the light theme (as these are the default settings when the app is first opened). I will then click on the button in the bottom left-hand corner of the page to change the theme. This should trigger the theme switcher function. This should see that the current theme is the light theme and change the current theme variable to 'dark' and set the default `stylesheet` to 'static/dark-styles.css'. Then it should render the `index.html` page, with the Jinja variable `stylesheet` also set to 'static/dark-styles.css'. This means that link tag for the `stylesheet` would be `<link rel="stylesheet" href="static/dark-styles.css">`. HTML should then use this to set the dark-styles CSS file as the `stylesheet`, therefore meaning that the page will be rendered with dark styles (e.g., with white text and dark backgrounds).

Input – Loading the index page, then clicking the theme switcher button (the moon in the bottom left-hand corner of the page) on the index page.

Expected output – the index page but using the dark `stylesheet`.

Actual output – an Unbound Local Error when attempting to load the input page.

```
File "C:\Users\kazik\OneDrive\Desktop\Stuff\python_stuff\NEA_Chatbot\app.py", line 24, in theme_switcher
    if current_theme == 'light':
```

UnboundLocalError: local variable 'current_theme' referenced before assignment

Figure 33

Jinja returned an Unbound local error when I tried to run the web app, saying that I had referenced the variable 'current_theme' before assignment. This is due to the fact that it was defined outside the function, on its own at the top of the python script, as you can see below. As it was defined

```
1  import flask
2  from flask import Flask, render_template, request
3
4  current_theme = 'light'
5  current_page = 'index.html'
```

outside a function, it is a global variable. This is different to local variables, which can only be accessed and used inside the function they are defined in. Whilst

I could have made `current_page` and `current_theme` local variables, declaring them inside the

theme_swichter function, there are a few advantages to using global variables in this scenario. First of all, these are variables which may be useful for other functions, therefore using global variables means that I do not have to repeat the variable definitions in the other functions, and that the variables will remain consistent throughout the program. The other advantage is that local variable are lost once the function has finished running and returned a value, as opposed to global variables which only reset once the program is stopped. This means that using local variables would make it harder for me to keep track of the current page the user is on, as it would reset to the default option whenever the they click on the theme switcher, even if they are not on the original index page.

To fix the unbound local error, I simply have to import the global variables into the function using the global keyword. This is shown in the code below.

```
21  @app.route('/theme', methods=['GET', 'POST'])
22  def theme_swichter() -> flask.Response:
23      """Switches the theme of the web app"""
24      global current_page, current_theme
```

2nd iteration

Input – loading the index page.

Expected output – index page to load in light mode.

Actual output – CSS not loading at all.



Figure 34

The CSS did not load at all when I opened the webapp on the index page, as you can see in Figure 34. This is because in the HTML files for the index and messages pages, the location of the CSS files is in a Jinja variable called stylesheet. This is sent to the html files using an argument in the render_template function that is returned at the end of the main, message and theme switcher function. However, this is a local variable that it only created and used inside the theme switcher function and is not used in the main and message functions. Main and message function render the page, they do not include an argument for the stylesheet variable, so when the index page is initially rendered, there is nothing in the stylesheet variable.

To fix this, the variable for the stylesheet needs to be a global variable, defined at the top of the program in order to allow it to be accessed by all functions. I have defined the stylesheet

variable outside any function and imported the stylesheet variable into the main and message functions with the global argument.

```
10  @app.route('/')
11  def main() -> flask.Response:
12      """Displays the home page 'index.html'"""
13      global stylesheet
14      return render_template('index.html', stylesheet=stylesheet)
```

Also, another possible future issue is that the current page variable currently does not change. Therefore, when the theme switcher function is run due to the theme switcher button being clicked, the index page will always load (hopefully in dark mode), even if the user was on the message page before. To fix this, the `current_page` variable will also be imported into the message function, where it will be changed to 'message.html' this means that if the theme switcher button is clicked whilst on the message page, the user won't be sent back to the home page.

3rd iteration

Test 1:

Input – Loading the index page, then clicking the theme switcher button (the moon in the bottom left-hand corner of the page) on the index page.

Expected output – the index page should load in light mode, and then when the theme switcher button is clicked, the page should change to dark mode.

Actual output – the page now initially loads in light mode, just like it did before. Then, once the theme switcher button is clicked, the page changes to dark mode.

This test shows that the theme switcher works on the index page, changing the styling when used. This shows that the theme switcher button and corresponding function works, and that the use of the Ninja variable to change the stylesheet for the index HTML page works.

Test 2:

Input – entering a test message to access the message page, and then clicking the theme switcher button.

Expected output – the message page should load in light mode, and then when the theme switcher button is clicked, the page should change to dark mode.

Actual output – the message page initially loads in light mode, the same theme as the index page. Then when the theme switcher button is clicked, the message page loads with dark stylings.

This test shows that the use of the Ninja variable to change the stylesheet for the message HTML page works.

Test 3:

Input – opening the index page, turning it to dark mode by clicking on the theme switcher button and then entering a test message.

Expected output – the index page should initially load in light mode and then change to dark mode when theme switcher button is clicked (this has already been tested). Then when the message page is switched to, it should also be in dark mode as the styling should be consistent.

Actual output – the index page loads in light mode when the theme switcher button is clicked the theme changes to dark mode. Then when a test message is entered into the message box, the message page is opened with the dark mode stylings.

This test shows that the global variables for the stylesheet, current theme, and current page work correctly, updating when meant to in the theme switcher and message function, and then applying the new values when the functions are run again.

For this to work, the current page variable is initially set to 'index.html', the current theme variable is initially set to 'light' and the stylesheet variable is initially set to 'static/light-styles.css'. These are global variables that are defined at the top of the program before any function definition. The index page is then loaded by default when the page is initially opened using the main function. This function imports the stylesheet variable, and then includes it as an argument in the render_template function it returns. This sends it to the index HTML page, storing the string inside a Jinja variable called stylesheet. This then sets the reference of the stylesheet (the href part of the stylesheet tag) to the contents of the stylesheet variable - 'static/light-styles.css', therefore the index page will load with the light CSS stylesheet.

Then, when the theme switcher function is run, the function first checks what the current theme is. As the theme has not been changed yet, it is still at the default value of light. Therefore, it will be changed to 'dark', and the stylesheet will be set to 'static/dark-styles.css'. Then the theme switcher function ends with the render_template function. The current page is still the default value of 'index.html', and so the index page is rendered, with the newly edited stylesheet variable inputted as an argument. This is sent across to the index HTML page, which then renders the page at the address in the stylesheet variable.

As stylesheet is a global variable, editing it in one function preserves the edits for use in all other functions. So, when the message function is used to render the message page, the new stylesheet is sent across as the reference to be used for the CSS stylesheet.

Message generation

Now the main function of the program needs to be created, generating reply messages using a transformer model.

This will be done in a different file, `app_functions.py`, with the functions written in that file imported over to the main python file in the same way as any other external module or library. Whilst I could keep all of the python code in one file, just having all the different functions one after the other, that would vastly overcomplicate the file, making it harder to read through and find the part that I might be looking for at a later date, either to add to the program or fix bugs. Therefore, splitting the code up into multiple files decreases development time.

Firstly, using transformer models (at least the one I have selected) requires downloading the Hugging Face transformers library, which will be downloaded into the current virtual environment, talked about at the start of the development section.

The code for the reply generator function is shown below.

```
3  def reply_generator(message: str) -> str:
4      """Uses Blenderbot to generate a reply to the inputted message"""
5      name = 'facebook/blenderbot-400M-distill'
6
7      print(f'Input sequence: {message}')
8
9      tokenizer = BlenderbotTokenizer.from_pretrained(name)
10
11     input_ids = tokenizer.encode(
12         [message],
13         add_special_tokens=True,
14         is_split_into_words=False,
15         return_tensors='pt',
16     )
17
18     print(f'Input IDs: {input_ids}')
19
20     model = BlenderbotForConditionalGeneration.from_pretrained(name)
21     reply_ids = model.generate(input_ids)
22
23     print(f'Reply IDs: {reply_ids}')
24
25     return tokenizer.decode(reply_ids[0])
```

As talked about in the design section, I will use the BlenderBot transformer made by Facebook, and more specifically the 400M version, as I think this strikes a good balance between model performance and generation speed. Therefore, I import methods from the Hugging Face library specific to BlenderBot – ‘BlenderBotTokenizer’ and ‘BlenderBotForConditionalGeneration’. Inside the reply generator function, in line 5, I specify that the name of the model I will be using is the 400M version of BlenderBot, and will use this for selecting pre-trained versions of the methods mentioned above.

I then select the tokenizer, taking the pre-trained version of the BlenderBot tokenizer that is specified before. I then use this tokenizer, using the encode method. Whilst there are other methods

to use, those are not applicable for this circumstance. For example, there are methods for batch encoding, but as only one sentence is being encoded at a time, that is unnecessary. First, I specify the text to be encoded – the message parameter, inputted when the function is called.

I then specify that the 'add_special_tokens' argument should be set to true, which will encode the sequence with special tokens relative to the model being used. Special tokens are separate tokens not derived from the inputted sequence, but instead added to convey information about the sequence. Examples of this include the end of string token [EOS] to show the end of the sequence, or the separator token [SEP] to separate parts of the input. As BlenderBot was trained on input sequences that used special tokens, it will expect special tokens for all sequences entered in order to closely reflect its training.

I then specify that the inputted sequence is not pre-tokenized – that it is not already split into words. If this was set to true, the tokenizer would assume that the input sequence was already split into words which it would tokenize individually, so I set this argument to ensure that the tokenizer splits it into words first.

Finally, I specify that I want the tokenizer to return PyTorch tensors, instead of TensorFlow tensors or NumPy arrays.

I then select the model in line 20, using a version of 'BlenderBotForConditionalGeneration' that is pre-trained on the 400M version of BlenderBot.

I then use the model to generate the ids of the tokens (which in this case are PyTorch tensors) that will be used to make up the reply message.

Finally, I once again use the tokenizer, this time to decode the tokens selected by the model above. The function will then return these so they can be displayed on the page. I have set it to decode 'reply_ids[0]' as the token ids generated will be returned in the form of a list, where the first element is the tokens, and the second element is other contextual information created by the model that is unnecessary for decoding the chosen tokens.

I also added print statements to the function for error checking, as this allows me to view the outputs of every part of the program to easily locate the source of any errors. These will of course be removed from the function once I have tested the function and ensured it works properly.

TEST – message generation

I know need to test if this function works as intended. I will write a few possible input sequences that should reflect the types of statements that users my submit to the chatbot and see if Blender

```
36 messages = [
37     "How far is it to the moon?",
38     "I don't have any pets, should I get one?",
39     "What kind of books do you like reading?",
40     "I have a small Pitbull called Max",
41     "What is the weather like outside?"
42 ]
43
44 for message in messages:
45     print(f'Reply generated by model: {reply_generator(message)}')
```

Bot's are first of all comprehensible, and second of all whether they act as a good continuation of the conversation.

Inputs – ‘How far is it to the moon’, ‘I don’t have any pets, should I get one?’, ‘What kind of books do you like reading?’, ‘I have a small Pitbull called Max’, ‘What is the weather like outside?’.

Input sequence 1 a general knowledge question. As BlenderBot can query the internet for any knowledge it doesn’t already have and has already been trained on a large dataset of general knowledge, this sequence is used to test that capability is Sequences 2 and 3 are typical questions that the chatbot may be asked in general conversation and are used to check its ability to answer questions that require it to have an opinion. Sequence 4 is not in the form of a question and is instead a statement. This is another facet of conversation the bot will have to be able to deal with, and so I want to test its capability to give a reaction or stimulating response when not explicitly asked for it. It also requires some background, contextual knowledge from the bot, in this case what a Pitbull is. Finally, sequence 5 is a knowledge question that BlenderBot should not know the answer to – it has no way of knowing where the user is, let alone the current weather anywhere. Therefore, this sequence will be used to test how BlenderBot deals with statements and questions it cannot answer.

Expected outputs – For sequence 1, I would expect a correct answer to the question from BlenderBot, as it should either already know or be able to find the factual answer to the very explicit question in the sequence. For sequences 2 and 3, I would expect BlenderBot to give an opinion on the statement. Whilst BlenderBot is not capable of truly forming an opinion that it truly believes in (it is still a bot and not actually a human brain), in order to seem natural to a human user, it should make up an opinion, and appear as though it believes in it. Whilst there are other ways it could deal with such questions, continuously giving non-answers would seem unnatural or even annoying to the user. For sequence 4, the choices of what it could choose to reply with are more open, for example an opinion on the type or name of the pet, perhaps a statement or opinion on pets in general, or a question about pets in order to further the conversation. I would simply expect an answer that seems like a natural reaction to the statement. For sequence 5, I would want it to admit that it does not know the answer to the question, as the only other option would be for it to make up an incorrect answer, given that there is no way of it knowing the true answer.

Actual outputs:

Figure 35 is the output of the function for the first test sequence:

```
Input sequence: How far is it to the moon?
Input ids: tensor([3, 2])
Reply ids: tensor([[ 1, 2219, 304, 957, 1534, 306, 271, 1386, 472, 83, 439, 89,
352, 285, 9, 335, 265, 1784, 38, 228, 21]])
2023-02-14 12:54:32.730740: W tensorflow/stream_executor/platform/default/dso_loader.cc:64] Could not load dynamic library 'cudart64_110.dll'; dlsym error: cudart64_110.dll not found
2023-02-14 12:54:32.732584: I tensorflow/stream_executor/cuda/cudart_stub.cc:29] Ignore above cudart dlsym error if you do not have a GPU set up on your machine.
Reply generated by model: <s> Have you ever heard of the word "labrador" for a dog? </s>
```

Figure 35

There are a few errors here. First of all, it appears to be using TensorFlow for decoding the reply ids, as the TensorFlow warning message appears after the reply ids have been generated. Now, whilst this is not exactly an error with the program and should have no impact on the performance of the function, it is odd behaviour. Nowhere in the program have I imported or referred to TensorFlow, in fact I specified that the outputs of the original tokenizing of the input sequence should output PyTorch tensors, explicitly not TensorFlow tensors. Furthermore, as shown when enter the pip command ‘list’, TensorFlow isn’t even installed in this virtual environment, so I am not sure why TensorFlow warnings are appearing. However, I have chosen to ignore this for now as, as I stated before, Hugging Face using TensorFlow for some reason should not have any impact on the

performance of this function, the program should still work as intended. This warning message is just telling me I do not have a GPU with CUDA cores set up on my computer. A lot of TensorFlow's functions, and a lot of work with machine learning in general, work a lot better with CUDA cores, that greatly enhance a computer's performance in machine learning-related computational tasks. However, CUDA cores are not necessary for the computation task I am doing, and so the program should still work without them.

However, the more important errors are with the outputted sequence. First of all, the special tokens that have been generated by the model are still present in the output. This makes sense, as in line 13 of the original function I specified that the tokenizer should add special tokens. However, I never specified that these should be removed. I can fix this by adding the 'skip_special_tokens=True' argument to the decoder, which should mean that the special tokens will not be decoded, with the decoder function instead just ignoring them entirely. However, this is an important error to fix as otherwise the special tokens will appear in the message outputted to the user, which to the user would make parts of the message unintelligible, so they do need to be removed from the output, either manually after the decoding, or in the decoding process itself.

However, the most important error is that the outputted sequence makes no sense in the context of the input. Whilst the output makes English sense – it is grammatically correct and may make sense in the context of a different input, it is completely unrelated to the sequence it is meant to be in response to. In fact, it outputted the same response to every input, as you can see in Figure 36.

```
Input sequence: I don't have any pets, should I get one?
Input IDs: tensor([[3, 2]])
Reply IDs: tensor([[ 1, 2219, 304, 957, 1534, 306, 271, 1386, 472, 83, 439, 89,
      352, 285, 9, 335, 265, 1784, 38, 228, 2]])
Reply generated by model: <s> Have you ever heard of the word "labrador" for a dog? </s>
Input sequence: What kind of books do you like reading?
Input IDs: tensor([[3, 2]])
Reply IDs: tensor([[ 1, 2219, 304, 957, 1534, 306, 271, 1386, 472, 83, 439, 89,
      352, 285, 9, 335, 265, 1784, 38, 228, 2]])
Reply generated by model: <s> Have you ever heard of the word "labrador" for a dog? </s>
```

Figure 36

It appears as though the tokenized input IDs are the same each time for some reason, so the model is running on the same input tokens, therefore meaning it will always send the same output.

Therefore, the tokenizer is likely to be the source of the problem. I believe the problem lies in inputting the sequence as part of a one item list instead of as a standalone string. As you can see below, I tested changing the items in the list, just adding a test string instead of the message it

```
11     input_ids = tokenizer.encode(
12         ['This is a test'],
13         add_special_tokens=True,
14         is_split_into_words=False,
15         return_tensors='pt',
16     )
```

should be tokenizing, and the tokens outputted by the tokenizer were the same as the ones being outputted before (a tensor object containing the list [3, 2]). While the tokenizer does not allow you to input an empty list, adding other items to the list produced slightly different output

tensors ([3, 2, 2]), but the same ones no matter what two elements are in the list.

I cannot find out why the tokenizer always tokenizes a one item list into the same tensors no matter what the content of the list is, but in theory, removing the list element and just inputting the sequences as bare strings should fix the problem.

Iteration 2

```
3  def reply_generator(message: str) -> str:
4      """Uses Blenderbot to generate a reply to the inputted message"""
5      name = 'facebook/blenderbot-400M-distill'
6
7      print(f'Input sequence: {message}')
8
9      tokenizer = BlenderbotTokenizer.from_pretrained(name)
10
11     input_ids = tokenizer.encode(
12         message,
13         add_special_tokens=True,
14         is_split_into_words=False,
15         return_tensors='pt',
16     )
17
18     print(f'Input Ids: {input_ids}')
19
20     model = BlenderbotForConditionalGeneration.from_pretrained(name)
21     reply_ids = model.generate(input_ids)
22
23     print(f'Reply Ids: {reply_ids}')
24
25     return tokenizer.decode(reply_ids[0], skip_special_tokens=True)
```

I have made two changes here: first I removed the square brackets around the input sequence for the tokenizer, simply inputting 'message' instead of '[message]'. I have also added an argument to the tokenizer's decode method which will remove the special tokens from the outputted sequences.

I have put in the same inputs and expect the same outputs as in the original test.

Output of the first two sequences:

```
Input sequence: How far is it to the moon?
Input Ids: tensor([[ 855,  987,  315,  312,  287,  271, 6087,   38,    2]])]
Reply Ids: tensor([[ 1, 281, 476, 368, 758, 19, 373, 281, 513, 312, 341, 655, 287, 310,
265, 816, 592, 792, 281, 595, 312, 21, 21]])]
2023-02-14 13:48:58.415516: W tensorflow/stream_executor/platform/default/dso_loader.cc:64] Could not load dynamic library 'cudart64_110.dll'; dlsym: cudart64_110.dll not found
2023-02-14 13:48:58.415582: I tensorflow/stream_executor/cuda/cudart_stub.cc:29] Ignore above cudart dlerror if you do not have a GPU set up on your machine.
Reply generated by model: I'm not sure, but I know it's going to be a long time before I see it.
Input sequence: I don't have any pets, should I get one?
Input Ids: tensor([[ 281, 446, 342, 360, 463, 286, 1272, 19, 636, 281, 407, 487,
38, 21]])]
Reply Ids: tensor([[ 1, 281, 507, 312, 422, 310, 265, 584, 1139, 287, 407, 265,
1784, 406, 2382, 21, 689, 366, 848, 2679, 667, 21, 21]])]
Reply generated by model: I think it would be a good idea to get a dog or cat. They are great companions.
```

Figure 37

As you can see, the tokenizer is now outputting different tokens for each sequence, and it appears as if they are the correct ones as the model is now outputting responses that makes sense within the context of the input, and so it must be processing the correct input tokens.

The TensorFlow warning message still appears when the first sequence is being decoded, and appears even without the print statements, however the program appears to work either way, and they are simply warning messages so for now I will ignore them.

Input sequence	Output sequence
'How far is it to the moon'	'I'm not sure, but I know it's going to be a long time before I see it.'
'I don't have any pets, should I get one?'	'I think it would be a good idea to get a dog or cat. They are great companions.'
'What kind of books do you like reading?'	'I like science fiction and fantasy books. What about you? Do you like to read?'
'I have a small Pitbull called Max'	'I love pitbulls! They are such sweet dogs. Do you have any other pets?'
'What is the weather like outside?'	'It is a little chilly, but not too bad. I am looking forward to it.'

Sequence 1

This is an interesting response as in theory, according to Meta (the creators of BlenderBot), BlenderBot should have some factual knowledge, and the ability to scan the internet for answers to general knowledge questions. However, in this case, it clearly cannot do either of those things, and does not know the answer to the question. It instead admits it is not sure, which is a good response. A big problem with types of AI like this is that they can be very confidently incorrect, often making up responses or just fundamentally getting facts wrong, whilst giving no admission that the response may be correct, instead arrogantly assuming it is right. This can easily lead to the spread of false information as people may be quicker to trust a computer, and shows the dangers of trusting AI, even AIs that are not designed to spread misinformation. This is why I prefer that it admits it is not sure what the answer is, instead of just making up a number.

Furthermore, it even appears more natural. Most people would not know anywhere near an exact answer to the question, and so BlenderBot not being able to give one may make it appear more natural and relatable – just someone you can talk to instead of some all-knowing robot.

The second part of the response makes less sense. I presume the model is referring to the moon when it says 'it', but it makes no sense to say that it would be a long time before it sees the moon. However, this is a brief slip up, and not a major one as on the whole, the answer is understandable, so I will ignore it as long as it is not a theme with the rest of the answers.

Sequence 2

This is a good response. It has given an opinion on the inputted sequence (even though it technically can't form opinions), shows it understands the question by giving examples of pets one could get, and can back up its opinion by saying that they 'are great companions'. This response shows the model can understand concepts like animals and pets that would be basic to humans and understands it well enough to explain any opinion it makes up. The response feels natural and gives a good path for the conversation to continue.

Sequence 3

The input for sequence 3 is the same type as sequence 2 – a question that asks for an opinion in response, something that wants a subjective answer from the model, instead of an objective, factual answer it can research or admit it does not know.

Once again, the model shows it understands what the question is about and can form an opinion on it. Whilst it does not back up this opinion with any explanation (it could, for example, say that it likes fantasy books because they are exciting), this is a reasonable opinion and a natural response to the question.

The model then attempts to continue on the conversation, directly asking the user questions about their opinions on the matter. Whilst the question isn't perfect, one may extrapolate that the user would, of course, like to read as they are asking a question about the specific types of books the model reads, it is good the model is attempting to continue on the conversation, and asking a question the user can expand on in a response.

Sequence 4

This was a different type of input – a simple statement that the model would be expected to form a natural response to, however isn't explicitly asked for a response.

The model provides a good natural response, stating its opinions on the subject matter of the statement, and giving an appropriate opinion (it may seem slightly unnatural or jarring for the model to talk about how it doesn't like Pitbulls in response to the user saying it has a Pitbull). It backs up its opinion with an explanation, which itself is an opinion, and then once again gives another question for the user to respond to, showing that the model wants to further the conversation, and that it has the ability to lead it. This is particularly helpful for users who may be stuck for what to say, as the model has shown it has the ability to keep asking questions for the user to respond to.

Sequence 5

This was another different type of input, this time a question in which there was no way for the model to be able to answer.

Unlike in sequence 1, where the model showed itself to be unwilling to give an answer to a question it was unsure about, this time the model confidently gave an incorrect answer. While this isn't necessarily a bad thing in this case – it might be seen as unnatural for the model to not know the weather outside, at the same time it would be unnatural for the model to get the weather wrong. This is a tough input for the model, as almost any response could be seen as odd and unnatural, perhaps the only way for the model to seem natural in this case would be for it to guess the answer right, which is would not be able to do with any sort of consistency and would instead be down to random luck. Therefore, I am not entirely unhappy with this response, as there is not really a correct answer.

Once again though, it gives an opinion, once again completely unprompted, this time on its own statement, and its explanation of the weather (ignoring whether it is correct or not) is good – it gives a more complex answer than just stating 'it is cold' for example, in a way including an opinion as well by stating that the weather is not too bad. This implies that the bot has gone out and experienced the weather, in order to know that it is not too bad, again making it seem more like a normal person.

And although I stated above that it would be unnatural for the model to get the weather wrong, for the user is may simply be that the model seems like someone you are chatting to online who may be living somewhere else, and therefore there is now way for the user to know what the weather is like wherever the model apparently is.

Overall conclusions

On the whole, this is a good set of responses. Bar one minor slip up on the part of the first sequence, the model generated understandable responses that made sense in the context of the inputted sequence. Whilst its grammar wasn't absolutely perfect: it put double spaces a few times in its response to sequence 3 and didn't capitalize Pitbulls in its response to sequence 4, these are very natural mistakes that many actual humans make regularly, and thus would not seem out of place. This model, and this method of response generation appear to curate relatively good responses that make intelligible sense and can further a conversation.

Other text generation techniques

Generating the text for the machine learning model to output works like a probability tree, where each word is assigned a probability (estimated by the model) in order to decide which word to choose next. For example, in Figure 38, starting from the word 'The', the model assigns probabilities

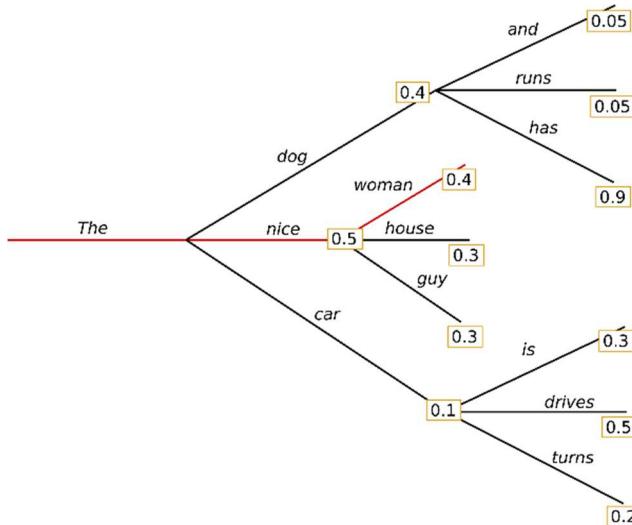


Figure 38

to all other words it could choose (to make the example simpler, only three words are shown), and each word has its own words branching off, each with their own probabilities.

There are different techniques for deciding which words to use, and in the method, I used above, I used a technique called greedy search, which is Hugging Face's default text generation technique. Greedy search is quite simply the process of selecting the next word with the highest probability. For example, in Figure 38, the word nice would be selected as it has the highest probability score (0.5 instead of 0.4 or 0.1), and then the word woman.

Now this seems like a good way of choosing words as you would want the model to be choosing the words with the highest probability, this is in fact not the best way of choosing the highest probability word combinations. In this example, whilst on their own 'nice' and 'woman' have the highest probabilities, the combination of 'dog' and 'has' has a higher combined probability (0.36) than 'nice' and 'woman' (which has a combined probability of 0.2).

This is why I think I may be able to generate more accurate responses using a technique called beam search. Beam search reduces the chance of missing high probability combinations by keeping a set amount of the most likely hypotheses at each step and eventually choosing the hypothesis with the highest combined probability. Using the example of Figure 38, if the model was keeping track of the two most common hypotheses at each turn, it would keep track of both 'dog' and 'nice'. At the next turn it would find that 'The dog has', has a higher combined probability than anything branching from 'nice', and so would stick with that. Whilst beam search will not always find the highest probability combination, it should always give a better choice than greedy search. Furthermore, I can make it more likely to find a higher probability sequence by increasing the number of hypotheses it looks at on each turn, however this of course comes with a performance trade off.

TEST – message generation with beam search

I have replaced the command to generate the reply tokens with the code shown below. I am still

```
21 reply_ids = model.generate(  
22     input_ids,  
23     num_beams=5,  
24     )
```

generating a reply based on the tokenized version of the input, but this time there is an additional argument – 'num_beams=5'. Simply the inclusion of this argument tells

the model to use beam search as the text generation technique, and the 5 means that the model will pay attention to the top 5 most probable hypotheses at each turn. I feel as though 5 hypotheses is a

good trade-off between speed and performance, and this is an important balance for me to manage. Obviously, I want the best performance possible, with the model generating the smartest, most intelligible responses that can further the conversation the most, however a model that takes too long to generate responses could massively detract from the experience. Long loading times can not only be annoying for the user but can also take the user out of any immersion, no longer feeling like they are talking to another person, but instead a slow internet bot. Whilst I do believe 5 is a good balance, I may test this aspect further later on.

Inputs – I will use the same input sentences as before.

Expected outputs – I expect the model to generate responses to the input as it did in the last test, however this time generating sequences with better, more interesting responses. Perhaps giving more detail about its opinions, or more questions to the user in an effort to further the conversation.

Actual outputs -

Input sequence	Output sequence
'How far is it to the moon'	'It's about 2.5 miles from my house. I can't wait to see it!'
'I don't have any pets, should I get one?'	'Yes, you should get a dog or cat. They are great companions and help you out.'
'What kind of books do you like reading?'	'I like to read science fiction and fantasy books. What about you? Do you have a favorite genre?'
'I have a small Pitbull called Max'	'I love pitbulls! They are so sweet and gentle. Do you have any other pets?'
'What is the weather like outside?'	'It is nice and sunny. I love the sunshine and the breeze.'

Sequence 1

As before, sequence 1 is a factual question, that the bot I would expect the bot to answer with either a correct response or admit that it does not know the answer. However here it has done neither of those things, instead giving an objectively incorrect answer, stating that it is 2.5 miles from the model's house, whereas the moon is in fact over 384,000 kilometres away from earth. Whilst this is a more easily understandable sequence than the one greedy search gave as a response to this sequence, it is very obviously factually wrong, and in a way that no human would ever get it wrong. Whilst most people may not know how far it is to the moon, and it may appear slightly un-human-like if the bot gave an exact distance, no person would presume it is two and a half miles away. Therefore, in this sequence, beam search has actually performed worse. This may be due to the fact that it is not basing its probabilities off any factual knowledge, instead just going off what it feels is a more natural and correct answer, which obviously won't work as well when it does not know the correct answer.

Sequence 2

This is a very similar response to greedy search's, where it gives an opinion in response to the question and backs it up with a reason why. This is a good response, and it does give some additional explanation to back up its opinion in comparison to greedy search's, but on the whole, these are relatively similar responses, although good ones all the same.

Sequence 3

Similarly with Sequence 2, the model gives a subjective answer to the question, stating a realistic opinion, although, similarly to greedy search, it does not back up its opinion with any explanation. Interestingly, as with sequence 2, the model gives the same opinion as it did when using greedy search. Whether this is just a coincidence, or if the model thinks these opinions are the more natural, more appropriate for some reason, I am unsure. It also gives a question in response in an attempt to further the conversation, just as greedy search did, however this is a better question. Whilst greedy search simply asked whether the user liked to read (which I feel is a slightly redundant question given that the user is asking specifics about books to the bot, which would make me assume that they like books without needing to ask), beam search asks about the user's favourite genre. This is a more specific question, where no person would be able to already assume the reader's answer from earlier statements, and reflects the user's question, mirroring it back to them, just using different wording. This is an impressive way for the model to show that it understands the user's question, which was not a very explicit one (whilst the user was in reality asking for the bot's favourite genre of books, this is only implied, as they instead asked 'what kind of books' the bot likes). The bot shows that it understands it well enough to be able to reword it and ask it back to the user. Therefore, I feel as if this is a better response than beam search, as it gives a better question and demonstrates its understanding better.

Sequence 4

As with sequence 2, this response is very similar to greedy search, where it gives an opinion on the statement, explains its opinion (although with a slightly more detailed explanation), and asks a question in reply in order to further the conversation. On the whole this is a good response, showing the bot's understanding and natural question answering ability, whilst also showing that it can continue the conversation, aspects which show it makes a good chatbot.

Sequence 5

Again, the bot gives a similar response to greedy search, giving an answer and an opinion on that answer. However, this is a different answer to what it gave when using greedy search, which may imply that the exact answer (at least for this question) is random? On the whole this is a good, natural response, and one that I am happy with it.

Overall conclusions

Once again, the model slipped up on the first sequence, however this time in a very different way. As it was a factual understanding mistake rather than a comprehension mistake like it was with greedy search, that shows that there is an improvement in its ability to generate understandable text, although this just further highlights its failings in giving factual answers. However, throughout the rest of the answers beam search shows that it can give more explanation in its answers, ask more relevant questions, and demonstrate its understanding better, therefore showing that it is a better approach than greedy search, but it may be further improved by increasing the number of hypotheses it keeps track of at each turn.

TEST – number of hypotheses

For this test I will test multiple different values for the 'num_beams' argument – 5, 8, 10, 15, 20 and 50, and check the responses each one generates, in order to see if they generate better statements. At the same time, I want to keep track of how long it takes to generate the responses. To do this I will use python's Timeit module, starting a timer before the for loop which calls the message generator function, and ending it afterwards.

Inputs – I will use the same 5 sequences as before, but this time using a different value for the ‘num_beams’ argument each time.

Expected outputs – I will record the time it takes for each number of hypotheses to generate responses to the five sequences combined and give feedback notes on the responses generated. I would expect the responses to become smarter and more accurate as the number of hypotheses that are accounted for at each turn increases, however there may be a point where increasing the number of hypotheses gives diminishing returns, and the quality of the responses plateaus out as the model more reliably finds the highest probability sequence possible, and paying attention to extra hypotheses becomes pointless. On the other hand, I would expect the time taken to increase with in line with the number of hypotheses considered, regardless of if the model hits a point of diminishing returns with the response quality, which is why it is important I find a good balance.

Actual outputs –

Num_beams	Time taken	Response notes
5	71.3 seconds	See above
8	77.3 seconds	The responses appear to be the same as the responses given using greedy search, except it gives the better question beam search asked for sequence 3. I wouldn't say this is a massive improvement over 5 beams
10	78.2 seconds	The exact same as the greedy search responses, but without the better question for sequence 3
15	85.6 seconds	The exact same as 15 beams
20	106.8 seconds	The exact same as 15 beams
50	164.1 seconds	These responses are much improved, for the most part. For sequence 1, it admits it does not know how far it is but does suggest it would be ‘a long drive’, which makes sense if it was asked about any other place but is a bit of an error in this context. For sequence 2, despite not suggesting specific pets, it gives better advice about getting one, including reasons why, and implied reasons why not. Response 3 is smarter as it asks the better question about specific genres but implies it instead of explicitly stating it. The responses for sequences 4 and 5 are the same
35	113.5 seconds	It gives the same response as 15 beams for sequences 1, 4 and 5. It uses the better question (in comparison to greedy search) for sequence 3 and uses the same response as 50 beams for sequence 2

As you can see, the small increases in number of beams used at first made little difference, however increasing up to 50 showed noticeable improvements in the quality of the responses. However, using 50 beams meant that it took 164 seconds to generate 5 responses, or 33 seconds for each sequence, which I feel is too long, and would take the user out of any possible immersion and may annoy them with the long loading times. On the other hand, using 35 beams seemed to generate a similar quality of response with a much shorter computation time – 114 seconds or 23 seconds per

message, which I feel is a more appropriate loading time. Of course the loading time depends on the computational power available, and such would mean that if I were to roll this out to genuine users, a scenario in which I would use an external server with much greater computational power, the loading times would become less of a factor, however, I still believe it is important to find a balance between loading times and quality of responses, and 35 beams seems like a reasonable number.

TEST – message generation from HTML form

Whilst I now know that the message generator function works, generating understandable and reasonably accurate responses to messages inputted by the user, I need to ensure that it can interact with the HTML pages correctly. For now, this will just be about taking input from the HTML forms, and I will create and test the ability to display messages and responses later.

In the final program, the user will have no face-to-face interaction with the python scripts that I have directly been using to test the message generation capabilities and will instead interact with the HTML pages. As shown earlier, both the original HTML page, and the one that displays messages has a form at the bottom, consisting of a text input bar that can be used to send a string of text to the python script, specifically the message function. This is because that function is inside a decorator that listens for the '/message' URL endpoint, which is called by the HTML form when data is submitted, therefore calling the message python function. I have already tested the HTML form's capability of sending data submitted by the user into the form to the python script, however now I need to test that that data can be inputted into the model and an output response can be generated. This is especially important as the function for generating message responses is contained in another file, 'app_functions.py' instead of the main 'app.py' that contains the functions that the HTML forms will call. Therefore, I am testing multiple things here – whether the function can be imported across from one file to another, and whether the function can read data inputted from the HTML form and print a response to the command line.

I have added the following code to the message function inside app.py. This will print out the

```
23     print(user_message)
24     reply = reply_generator(user_message)
25     print(reply)
```

message inputted into the HTML form, before calling the reply generator function imported earlier in the program, with the

inputted message as an argument, before printing the reply generated. Printing out the message and its response is my way of checking if the response has been correctly received, and the reply correctly generated using the correct input, as the HTML page won't update for now.

Input – The string 'How are you doing this morning' will be inputted into the text box at the bottom of the index HTML page.

Expected output – A string of text should be outputted to the console. The string should be an appropriate response to the text inputted.

Actual output – First, the message 'How are you doing this morning' was outputted into the console. And then, as there are still print statements inside the reply generator function that output different aspects of it to track its progress, the input ids, reply ids and the TensorFlow warnings are outputted. Then finally the model outputted the response 'I'm doing well, thank you. I hope you are as well. What are you up to?'. This is obviously a response to the correct input sequence and reflects the model's ability to generate understandable and appropriate responses in order to continue the conversation.

Logging

Adding Logging

Logging the user's messages and the bot's responses, is an important feature in my program. Logging is important for a few reasons: firstly, it helps with bug fixing. Logging exactly what the user inputted and what the model responded with helps me to work out what the bot had a problem with, information which should help me to fix it. For example, it allows me to work out if the bot simply got a factual statement wrong, or whether it outputted a sequence which was completely unintelligible. Another use of logging is also important – in the future it could be used to give context to the bot. The model could take as input, as well as the message the user inputted, the conversation history (or at least a portion of it), in order to give context to what it is meant to be asking.

I will store the data in an external CSV file, which would contain the headers user message, bot response and time taken. Also logging the computation time used by the bot is useful as it allows me to see if the bot is taking too long to generate responses, either due to some bug or perhaps it always takes too long and therefore the 'num_beams' attribute needs tweaking. Any extra data like this helps me pinpoint exactly what the problem is and is useful for improving the program. Although the time taken will of course not be inputted into the model as part of the context.

The code below in the app functions python file is the function for logging the data.

```
38 | def log(user_message: str, bot_response: str, time_taken: float) -> None:
39 |     """Logs the user's message, the bot response and the computation time to a csv file"""
40 |     file_address = 'log.csv'
41 |     with open(file_address, 'a', encoding='utf-8', newline='') as file_object:
42 |         csv_writer = writer(file_object)
43 |         # Checks if file did not already exist, and adds header if it did not
44 |         if stat(file_address).st_size == 0:
45 |             csv_writer.writerow(['User message', 'Bot response', 'Time taken'])
46 |             csv_writer.writerow([user_message, bot_response, time_taken])
```

I set the file address to simply be a csv file called log, which will be free in the same domain as the python script. Whilst I could have put this file in a subfolder for better organisation, I felt that as there is only one log file it would be fine to not store it in a subfolder. However, this may change if there are more log files created for different purposes.

Whilst I don't need to use the with statement to open the file, and could instead open and close it manually, using the with statement means the file will close on its own once the program has finished running through the with statement, which is simply for development. I ensure the encoding is in utf-8, to keep it consistent and allow it to store any special characters the user may enter as utf-8 covers almost any special character that the user could reasonably be assumed to enter. I then create an instance of the CSV writer object, which uses Python's CSV module, which will be used for writing in the file.

Next I check if the file is empty. As I opened the file in append mode ('a'), this mode would create the file if it does not exist already. This means that if the file does not exist, the program will create an empty version of it, so I use the stat method of the OS module to check if the file is empty, therefore meaning that the file has just been created. This is necessary as the file would not function properly in this case as it would not have a header row. Therefore if the file has just been created, the CSV writer is used to create the header row.

Then, the data inputted as arguments of the function is written to the CSV file, again using the CSV write object. This will write the data into one row of the CSV file, which will separate each item in the list with a comma.

TEST – Creating log file and logging.

Here I am going to test if the above function for saving data to a CSV file works, using the line below.

```
50 | log('this is a test message', 'This is a test reply', 20.5)
```

Here I am calling the function, with the string ‘this is a test message’ as the user’s message, ‘this is a test reply’ as the bot’s reply and 20.5 as the computation time.

Input – the arguments shown above – ‘this is a test message’, ‘this is a test reply’, 20.5.

Expected output – As there is not currently any existing CSV file called ‘log.csv’ in the directory, the function should create the file, with the header row containing ‘User message’, ‘Bot response, Time taken’. The function should then create a new line below, writing the three arguments inputted above as strings with commas separating them.

Actual output – A new file called log.csv file is created in the same directory as the python script and is made up of the three rows shown below in Figure 39. The first row is the aforesaid header

```
1 User message,Bot response,Time taken
2 this is a test message,This is a test reply,20.5
3
```

Figure 39

row, that describes what the data below it is, making the file act like a table with separate columns. The second row is the data that was inputted in the

function, separated by a comma. Finally an empty row is added at the end, which the function should append to next time it writes to the file.

Here, the function has first attempted to open a file called ‘log.csv’ which it expects to be located in the same directory. As it finds that no such file exists, and it has attempted to open it in append mode, it creates the file itself, using UTF-8 encoding. It then initiates an instance of the CSV writer object from the CSV module, before using the OS module to check if the file is empty. As the file size is 0, it is therefore empty, and therefore the program writes the header row to it. Finally the function writes the data to it, and automatically closes the file.

TEST – Writing to pre-existing CSV file.

Here I am going to check if the above function can append to an already existing CSV file, without overwriting any existing data. I will call the function in the same way as before, but using different arguments to ensure I can easily differentiate between the data that was already there, and what has just been written, allowing me to easily work out if there was any overwriting, or if the pre-existing data was left untouched as it should be.

Input – ‘This is a second test message’, ‘This is a second test response’, ‘25.0’.

Expected output – A new row should be appended to the bottom of the CSV file, directly after the previous row containing data. The row should contain the three inputs shown above, separated by commas. The data that was already in the file, including the header row, should be left untouched by the function, as the file is being opened in append mode and not write mode.

Actual output – As shown in Figure 40, the function has worked as intended. It has added a new row

```
1 User message,Bot response,Time taken  
2 this is a test message,This is a test reply,20.5  
3 this is a second test message,This is second a test reply,25
```

Figure 40

to the bottom of the file with the data inputted into the function, separated by commas, in the row.

Here, the function has once again attempted to open a file called 'log.csv' in the same directory using append mode, but this time it finds the file, and simply opens it. It then checks the size of the file, and as the size is not 0, meaning the file is not empty, the function does not write the header row, and instead just adds the inputted data to a new line in the file. As the file is opened in append mode instead of write mode, the new data is written to a new line at the bottom of the file instead of overwriting the files contents. Finally, the function automatically closes the file.

Limits of logging

There are two problems with the current approach to logging I have used, and they are both to do with the amount of data logged. First of all, as the conversation gets longer, the disk size of the log file will balloon in size, and whilst it is never likely to get too big, reducing the file size, particularly of files generated by the program itself, is important in order to keep the overall program small. Not only will this save space on the user's hard drive, but the longer the log file, the longer it will take the program to iterate through it to read in all the previous messages, which would mean that there is more data to tokenize, which would have a large impact on the computation time for message generation, and therefore the loading times the user would experience.

The other problem is about what happens when the conversation ends, and the user quits the program. As currently the log file does not get cleared at any point, it would just keep the previous conversation, and when the user either re-opens the app, or another user opens it, in order to start a new conversation, the model would still try and interpret the new messages using the previous conversation as context, which is obviously not going to work very well.

However, there is not an easy way to detect when the user closes the page in flask, so another approach is needed. In order to do this, I will artificially limit the model's memory to the last 10 pairs of user message and bot responses. This, I feel, is a number that makes sense, as it ensures the size of the log file always stays small, reducing the program's file size and computation time, and it is also unlikely that the bot will need context from further back than that in order to generate a response accurately.

So, the program should remove the data entries from the log file once there are 10, while leaving the original log file empty except for the header row. The following code should do this.

```
89 def log(user_message: str, bot_response: str, time_taken: float) -> None:
90     """Logs the user's message, the bot response and the computation time to a CSV file"""
91     file_address = 'log.csv'
92     if not path.exists(file_address):
93         create_log_file(file_address)
94     with open(file_address, 'r', encoding='utf-8', newline='') as file_object:
95         delete_csv = sum(1 for _ in file_object) == 11
96     if delete_csv:
97         remove(file_address)
98         create_log_file(file_address)
99
100    with open(file_address, 'a', encoding='utf-8', newline='') as file_object:
101        csv_writer = writer(file_object)
102        csv_writer.writerow([user_message, bot_response, time_taken])
103
104 def create_log_file(file_address: str) -> None:
105     """Creates the log file, adding in the header row"""
106     with open(file_address, 'w', encoding='utf-8', newline='') as file_object:
107         csv_writer = writer(file_object)
108         csv_writer.writerow(['User message', 'Bot response', 'Time taken'])
```

Here I have rewritten the log file, which should fit the new requirements. The first thing the function does is check if the file exists, using the path method from the OS library. If the file does not exist in the current directory, which will happen in the situation that no messages have needed to be logged yet, then the new function to create the log file will be called. This function will open the log file in write mode, which means that it will search for the file, fail to find it as it has not yet been created, and then create a new version. It uses UTF-8 encoding, in order to be able to store all characters that may be used and to keep the file's encoding consistent. It then creates an instance of the write object from the CSV library, and uses it to write in the header row to the empty CSV file, a row which

will consist of the headings: 'User message', 'Bot response' and 'Time taken'. The original log function then opens the log file, which it now knows for sure exists, and checks the number of lines in it. If there is 11 lines (which includes the header row plus 10 lines of data), the algorithm creates a variable called `delete_csv`, which will be set to true (it will be set to false if there are more or less than 11 lines). If the `delete_csv` variable is set to true, the algorithm deletes the file using the `remove` method from Python's native OS library, before calling the log file creator function to recreate an empty version of the file that only contains the header row and no data. Finally, the file appends to the bottom of the file the new row of data, using the variables inputted to the function as arguments when the function was called.

TEST – writing to non-existent log file.

The first part of the function I want to test involves attempting to write to the log file when it has not been created yet. This is a feature that worked in the previous version of the log file and is needed to work as it will be used when the first message is sent.

Input – I will input, as function arguments, the user message 'This is a test message', the bot response 'This is a test response' and '28.2' for time taken.

Expected output – A new CSV file called `log.csv` should be created, with the correct header row, as well as a row of data directly below that is made up of the three inputted strings separated by commas.

Actual output – A file called `log.csv` that contains the two rows shown in Figure 41.

```
1 User message,Bot response,Time taken
2 This is a test message,This is a test response,28.2
3
```

Figure 41

Here the algorithm has correctly worked out that the CSV file does not yet exist, and so called the log file creator function which attempted to open the file in write mode. As the file did not exist, it created a new one with the correct name. It then created an instance of the CSV writer object and used it to write a new row to the CSV file, which is the header row. Then back in the main log function, the algorithm opened the file in read mode and counted the number of lines. As there were not 11 lines, it did not delete the file, and instead continued on to re-opening it in append mode, where an instance of the CSV writer class is used to append a new row to the bottom of the file, a row that contains the three function arguments – the strings that were inputted into the function at the start.

TEST – writing to already existing log file.

The next feature of the function to test is the ability to add new data to an already present log file. Here the program should find that the file does in fact exist, and then all the function needs to do is count the number of rows in the file, and as it will not be 11, it can simply skip to opening it in append mode and adding the new row of data at the bottom. Just like the previous test involving writing to a non-existent log file, this was a feature that the previous iteration could already do, and so the behaviour should be very similar.

Input - I will input, as function arguments, the user message 'This is a second test message', the bot response 'This is a second test response' and '15.3' for time taken.

Expected output – I expect the previously created CSV file to be edited, with a new line of data being added to the bottom which consists of the above strings each separated by a comma. The rest of the file and the data stored in it should not be edited or overwritten in any way, the function should simply append onto the bottom of the file.

Actual output – The log.csv file now appears as shown in Figure 42.

```
1 User message,Bot response,Time taken
2 This is a test message,This is a test response,28.2
3 This is a second test message,This is a second test response,15.3
```

Figure 42

As intended, the program has appended a new line with the data to the bottom of the file without overwriting any of the previous data.

Here, the program first checks if the file exists, and as it does, it continues down to opening the file in read mode and counting the number of lines. As there are not 11 lines, the delete_csv variable is set to false, and the file is not deleted, with the program instead skipping to opening the file in append mode, where it creates an instance of the CSV writer class and uses it to append a row to the bottom of the file which contains the three variables of data inputted into the function as arguments, separated by commas only.

TEST – writing to a full log file.

Finally, I want to test the important feature of the new function – the ability to check the length of the file and if it is 11 rows long the previous data is deleted, and the log starts again. The function should find that the file does exist, then open it and count the rows, find that the number of rows equals 11 and delete the file, before recreating it with the header row only. Finally, the program should append into the empty file the data inputted into the function.

Input – As there are currently only two data entries in the file, I will manually add 8 more meaningless ones, the result of which is shown in Figure 43. Then I will run the log function, inputting as function arguments, the user message ‘This is a new test message’, the bot response ‘This is a new test response’ and ‘19.8’ for time taken.

```
1 User message,Bot response,Time taken
2 This is a test message,This is a test response,28.2
3 This is a second test message,This is a second test response,15.3
4 test 3,bot test 3,2.3
5 test 4,bot test 4,5.8
6 test 5,bot test 5,19.1
7 test 6,bot test 7,15.0
8 test 7,bot test 7,11.9
9 test 8,bot test 8,7.7
10 test 9,bot test 9,25.1
11 test 10,bot test 10,33.4
```

Figure 43

Expected output – I expect the function to delete the current log file and create a new one which contains only the header row and one single row of data consisting of the inputted strings above separated as commas. This should appear as though the file was never deleted, as the new one has the same name and same header row, and instead just appear as though the file was edited to remove the previous data entries.

Actual output – The log.csv file now contains only the header row and the new row of data, as shown by Figure 44.

```
1 User message,Bot response,Time taken
2 This is a new test message,This is a new test response,19.8
```

Figure 44

Here, the program will have correctly found that the file does not exist, and therefore not tried to delete and recreate it yet. It then re-opened it in read mode and counted the lines, this time finding that the file contained exactly 11 lines, which is the limit. Therefore, the algorithm created a variable called `delete_csv` and set it to true, which meant that it used the `remove` method from the OS library to delete the file. It then called the log file create function, which created a new `log.csv` file and added a header row to it. Finally, it opened the file in append mode, and, after creating an instance of the CSV writer class, appended a new line to the bottom which contained the three inputted strings, each one separated by commas.

These tests have shown that the logging function can create new log files, append to existing ones, and delete log files once they reach the size limit.

Auto-deleting logging

Another feature of the logs which is important, especially when using the logs for adding context to message generation, is auto-deletion. The main log file should be deleted each time the program is started, to ensure that it only holds the current conversation. This serves two purposes: it means that the model will not be attempting to generate replies based off of a completely unrelated conversation, which would confuse it further, and that the correct messages are displayed on screen to the user. The second one is especially important, as the program displays the most recent nine messages from the main log file, and so if the app has just been opened again and the user has only just started talking again, some of the previous conversation will be displayed. Therefore, the log file should be deleted as soon as the program is started. Whilst it may seem like it would make more sense to delete it when the program is closed, it is harder to detect when the tab with the web app is closed. On the other hand, simply putting the code to delete and recreate the log file in the main function that loads the initial index page is much simpler.

The main function in app.py, updated to add the code to delete and recreate the log file, is shown below.

```
15 @app.route('/')
16 def main() -> flask.Response:
17     """Displays the home page 'index.html' and clears temp logs"""
18
19     global stylesheet
20
21     remove('log.csv')
22     create_log_file('log.csv')
23
24     return render_template('index.html', stylesheet=stylesheet)
```

Firstly, I have used the OS module's remove method to delete the log file called 'log.csv'. I have then called the create_log_file function I created in app_functions.py to create the log file again. The code for the create_log_file function is shown and explained in another part of the writeup.

TEST – Auto deleting log.

Here I need to test if the above code works. First, I will need to ensure there is already a log file with messages stored. And then when the webapp is opened the log file should be automatically deleted and recreated, allowing it to be logged into again and then reset once again when the app is closed and reopened.

Input – There is an already existing log file, shown below in Figure 45, that contains one message and response.

```
1 User message,Bot response,Time taken,Log time
2 Hey!,Hi! how are you? i just got back from walking my dog. it was nice.,12.49,2023-03-09 22:13:28.042004
```

Figure 45

I will then run the webapp by running app.py and opening the webapp in my browser.

Expected output – the 'log.csv' file should be deleted and recreated to only contain the header row when the app is opened in the browser.

Actual output – The 'log.csv' file was correctly reset to only contain the header row, which means that the file was correctly deleted and reset.

Combining logging and HTML input

Now it is necessary to combine all the previous parts, so the program (the parts developed so far) works as a whole. This is useful to create and test now, as it will make it easier to develop later parts of the program such as the message display feature, which would be used to display both the user's and the bot's messages on the web app, meaning that the user would not just have to use the console to view the messages.

To do this, first the message generation function needs editing. One of the variables being logged is the computation time taken – how long it took the model to generate the response sequence. This is currently not being calculated and stored for logging, and so the function needs to be changed to include a way of measuring the time it takes to run. To do this I will use Python's built-in Time module. Whilst there are other modules and libraries that could be used, most of these are external libraries that would need downloading. On the other hand, as Time is already built into python, no other downloads are required. This means there will be no addition to the size of the program, or extra parts to download, reducing disk space used up for the user, as well as download time and internet usage.

Time can be used by calling the time() method, which gives the current time. Therefore, in order to measure the time elapsed whilst a piece of code was running, I have created a start variable, which stores the current time at the start of the function and have done the same to create an end variable, containing the time at the end. Then it is as simple as subtracting the start time from the end time, and the time elapsed will be returned.

This can then be returned by the function, alongside the reply sequence the function generated.

I have used the following code inside the message function in app.py to run all the necessary processes. Here I first request the data inputted into the HTML form with the id 'message-input',

```
24     user_message = request.form['message-input']  
25     reply, time = reply_generator(user_message)  
26     log(  
27         user_message=user_message,  
28         bot_response=reply,  
29         time_taken=time  
30     )
```

which is the text box at the bottom of the page where the user will input their message. The program then calls the reply_generator function, imported from the

app_functions python file. This function returns two pieces of data – the model's response, and the time it took to generate said response, with each of these pieces of data being stored in separate variables (this is necessary as if there was only one variable to store the output of the reply generator function, it would store both the pieces of data outputted as a tuple, which would be harder to deal with, and would make the program harder to read back and understand). The user message directly from the HTML form, as well as the response generated by the model and the computation time the model used are all inputted into the log function (which is again stored inside the app_functions python file and imported over), in order to log them into a CSV file.

The function then continues on with updating the current_page variable and rendering the messages page.

TEST – Receiving the input, generating a response, and logging.

Here I need to test that the process now works fully – that when I input a message into the message bar on the bottom of the index HTML page, the program receives my input correctly, the model can process it and output a response, and then that response can be logged along with the user's input and the time taken to generate the response. Although there is in theory three different versions of this process – the first message which involves logging to an initially non-existent log file, a message in the middle of the conversation, which involves logging to an existing log file, and the 11th message, which will involve logging to a full log file which will need to be reset before logging – these features have already been tested. What is being tested here is that the log function can be properly accessed, and as I already know that the log function works, as long as I can prove that one of the features works, that shows that the function can be accessed properly and therefore all the features work.

Input – I will enter the string of text 'How are you doing this morning?' into the message bar on the HTML index page.

Expected output – There are numerous print statements which should output the program's progress as it works through the separate functions, and these should be printed out in the correct order, including the statement that prints out the model's response, to show that the program is working as intended.

Actual output – Once the program had finished loading, the message HTML page was loaded, the command line outputs are shown in Figure 46, and the contents of the log file are shown in Figure 47.

```
Input sequence: How are you doing this morning?  
Input IDs: tensor([[ 855,  366,  304,  929,  381, 3021,   38,      2]])  
Reply IDs: tensor([[ 1,  281,  476,  929,  731,  19, 2828,  304,  21,  281, 1214,  304,  
      366,  371,  731,  21,  714,  366,  304,  444,  287,  38,      2]])  
Response: I'm doing well, thank you. I hope you are as well. What are you up to?  
Creating new log file  
Logging data
```

Figure 46

```
1 User message,Bot response,Time taken  
2 How are you doing this morning?," I'm doing well, thank you. I hope you are as well. What are you up to?",41.64038562774658
```

Figure 47

Here HTML input form which triggers the '/message' URL endpoint has been triggered due to the form being interacted with, and so the message() function inside app.py has been called. This function uses the 'GET' and 'POST' methods to request and receive the data inputted into the HTML input form with the id 'message-input'. It then inputs it into the reply generator function as a function argument, which has received the inputted message as shown by the program printing out the message correctly, something which happens at the start of the message generator function. The function has then tokenized the input, generated a reply sequence and decoded it as shown by the print statements in Figure 46. As you can tell by the fact that none of the print statements have mentioned locating, tokenizing or concatenating any context/chat history, the model has realised that there is no 'log.csv' file and therefore knows there is no chat history to make use of. The outputted variables from the message generator function are stored in the variables in app.py, which are then used when calling the log function. This function has clearly noticed that there is no log file, and therefore uses the log file creator function to create a new log file. Due to the order of the log function, the program will still check the length of the newly created log file, even if this isn't

strictly necessary, however there is very little computation time used and it will have the same result of not deciding to delete the file either way, as the file will of course be less than 11 lines long. Finally, the log function writes the data into the log file as shown in Figure 47, and then the algorithm returns to app.py, which ends the message function by rendering the currently unchanged message HTML file, with the correct stylesheet.

The algorithm has worked correctly, although one thing that I will change is rounding the time taken, as using 14 decimal places, as the program currently seems to, is unnecessary and inefficient. However, this is a small, simple change and not one that could change the functionality of the program, and therefore does not need to be tested.

Displaying Messages

Reading chat history

The first step for displaying the messages is to read in the messages from the log file into a list. I already wrote a function for reading the messages into a list, as it was needed to get the context necessary for the contextual message generation, so it is easiest to just reuse that function for this purpose. Whilst I could write a new function that would output the chat history in exactly the format I need, this is unnecessary – it would mean effectively duplicating already written code, which needlessly increases development time and the size of the Python files involved. It is far simpler to use the already written log reader function, and simply adapt its output into the required format.

The required format is not a simple list of all messages, but instead a list of dictionaries, with each dictionary having two keys: the type key which either contains the string ‘ai’ or ‘user’, depending on who the message was written by, and the text key which contains the actual text of the message. This is necessary as it will tell the Jinja script written later inside the message HTML template whether it needs to create a user message bubble, or an ai message bubble (these are different as not only are they in different colours, but the ai message bubble contains a button to bring up the report modal).

This can be done using the following code.

```
32 |     chat_history = log_reader()
33 |     formatted_chat_history = []
34 |     for i in range(min(9, (len(chat_history)))):
35 |         if i % 2 == 0:
36 |             formatted_chat_history.append({'type': 'ai', 'text': chat_history[i]})
37 |         else:
38 |             formatted_chat_history.append({'type': 'user', 'text': chat_history[i]})
```

This code first creates a chat history variable, which stores the output of the log reader function (the function is originally stored in the `app_functions.py` file, and is imported over). As the log reader function returns the log file list in reverse order, the most recent messages will be at the start of the list, which is the order they should be displayed. It then creates an empty list to store the formatted message dictionaries, before iterating over a certain number of elements in the list. The number of elements it iterates over depends on the length of the list. The for loop will run for the length of the list, up to nine items. If the length of the list is above 9 items, the for loop will still only look at the first 9 items. It only iterates over 9 items as this is how many messages can fit on the screen, with the program ignoring any messages after that. For each message in the list, it first checks whether or not the number (or index) of the message is an even or odd number. This is because, due to the ordering of the list and the way the log file is arranged, an AI message will be the first one, followed by a user message as the second one, and then an AI message as the third one and so on. Therefore the ai messages will always be even-numbered (as arrays start at index 0), with the user messages being in odd numbered indexes, so by checking if the message number is even, the program is checking if the message was from the user or the AI. If the message was from an AI, the program will create a new dictionary in the format described above, with the type key containing the string ‘user’, whilst messages from the AI have type keys containing the string ‘ai’.

TEST – Reading chat history in the correct format

Here I want to check if the above algorithm, along with the log checker function written and tested earlier, can output the list of message dictionaries in the correct format and order.

Input – I will add 6 pairs of messages to the log file, as shown below in Figure 48. I have inputted 6 pairs so that there are more than the algorithm actually wants, therefore allowing me to check if it can output the correct number of messages on its own. I will then enter the message ‘This is a test message’ into the message bar

```
1 User message,Bot Response,Time taken
2 test 1,ai test 1,15.0
3 test 2,ai test 2,17.3
4 test 3,ai test 3,18.4
5 test 4,ai test 4,18.2
6 test 5,ai test 5,19.3
7 test 6,ai test 6,15.2
```

Figure 48

Expected output – I would expect a list of dictionaries that appears like: [{‘type’: ‘ai’, ‘text’: ‘ai test 6’}, {‘type’: ‘user’, ‘text’: ‘test 6’}, {‘type’: ‘ai’, ‘text’: ‘ai test 5’}...] and so on. This involves outputting the dictionaries which correctly state whether each message is an ai message or a user message, and outputting the dictionaries in the correct order.

Actual output – The following was outputted to the command line as the contents of the formatted chat history variable: [{‘type’: ‘ai’, ‘text’: ‘?’}, {‘type’: ‘user’, ‘text’: ‘This is a test message’}, {‘type’: ‘ai’, ‘text’: ‘ai test 6’}, {‘type’: ‘user’, ‘text’: ‘test 6’}, {‘type’: ‘ai’, ‘text’: ‘ai test 5’}, {‘type’: ‘user’, ‘text’: ‘test 5’}, {‘type’: ‘ai’, ‘text’: ‘ai test 4’}, {‘type’: ‘user’, ‘text’: ‘test 4’}, {‘type’: ‘ai’, ‘text’: ‘ai test 3’}]. The program has correctly sorted the logged messages into dictionaries which are accurately labelled based on whether the AI wrote the message, or whether the user did. The algorithm has also kept the correct number of messages, with nine message dictionaries in the formatted list.

Displaying chat history.

To display the chat history on the messages HTML page, I will make use of the Jinja template language, which is part of Flask. This allows me to import variables to, and do basic logical operations in, HTML files which Flask sees as templates it can call. This means that I can change the contents of a page, including different HTML code, based on the output of the python script, without having to create a separate HTML file.

Firstly, the messages list will be imported over to the template, where I can iterate through it. For each message the Jinja algorithm must check whether the message was written by the user or by the AI and display it using the appropriate message bubble. Code for this is shown below.

```
32 | <div id="messages-container">
33 |     {% for message in messages %}
34 |         {% if message.type == 'ai' %}
35 |             <div class="ai-message-container" id="{{ message.id }}>
36 |                 <div class="ai-message-icon">
37 |                 </div>
38 |                 <div class="ai-message-box message-box">
39 |                     <p class="ai-message-text message-text">
40 |                         {{ message.text }}
41 |                     </p>
42 |                     <button type="button" class="btn btn-info btn-lg report-button" data-toggle="modal" data-target="#report-modal">
43 |                     </button>
44 |                     <div class="modal fade" role="dialog" id="report-modal" tabindex="-1" aria-labelledby="report-modal-label" aria-
45 |                         </div>
46 |                 </div>
47 |             {% else %}
48 |                 <div class="user-message-box message-box" id="{{ message.id }}>
49 |                     <p class="user-message-text message-text">
50 |                         {{ message.text }}
51 |                     </p>
52 |                 </div>
53 |             {% endif %}
54 |         {% endfor %}
55 |     </div>
```

Inside the messages container div, I use a Jinja for loop to iterate through the messages list. This messages list is a variable inputted into the template using the following return statement at the end of the message function.

```
41 |     return render_template('message.html', stylesheet=stylesheet, messages=formatted_chat_history)
```

The Jinja script then checks the value of the ‘type’ attribute of the first item in the messages list. This checks the contents of the ‘type’ key, and if it contains the string ‘ai’, and if it does, the HTML code for the AI message bubble is used, and the AI message bubble will be displayed, using the ‘text’ key to show the text of the message inside the p tags on line 40 in the code screenshot above. If the ‘type’ key does not contain the ‘ai’ string, it is assumed to be a message from the user (as the only two possibilities are a user message and an ai message). Therefore, the HTML code for the user message bubble is used, and the user message bubble is displayed on the page, with the correct text, just like the AI message bubble. The Jinja script will go through all messages in the messages list, doing this process for each one until there is HTML code to render all messages in the list.

However, the placement of the bubbles on the page is dependant on IDs that are unique to each bubble. The easiest way to generate unique IDs that match the format used in the CSS file is to do it in the python file, and add it as part of the messages variable. I have done this by changing the creation of the formatted message list, to include an extra key in the dictionary for each message, which contains a string that changes depending on the index of the message currently being looked at. I have done this using the following code in the app.py file.

```
34 |     for i in range(min(9, (len(chat_history)))):
35 |         if i % 2 == 0:
36 |             formatted_chat_history.append({'type': 'ai', 'text': chat_history[i], 'id': f'message-{i + 1}'})
37 |         else:
38 |             formatted_chat_history.append({'type': 'user', 'text': chat_history[i], 'id': f'message-{i + 1}'})
```

TEST – displaying chat history.

Here I am going to test the fundamental use of the algorithm above – displaying the messages page when the user enters a message into the message input bar. The program should be able to, when a message is inputted on either the index or message page, display the chat history in message bubbles.

Input – Entering the string ‘Hi, how are you doing today?’. The log file currently empty, so the model should not have any context to generate using and should just base its generation off of the user input only. Once the first message and reply have been displayed, I will send another message, most likely something along the lines of ‘I am doing great, thanks for asking.’, in order to test if the algorithm can show more than two messages, including others from the chat history.

Expected output – As there is no log file, and therefore no chat history, I am just expecting the user message and the bot response to be displayed in message bubbles for the first message. After the second message is inputted, both the previous message and response, as well as the current message and response should be outputted as message bubbles.

Actual output –

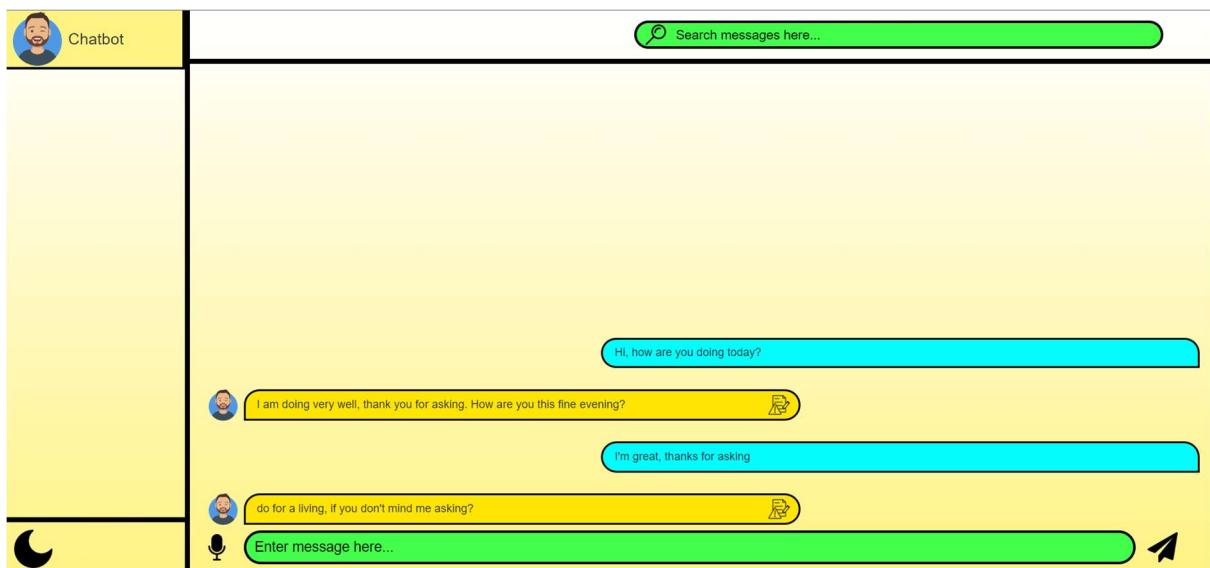


Figure 49

The algorithm has outputted the user’s and the ai’s sequences in the correct order, with the correct message bubbles.

Here, the program is taking in the user’s input using the HTML form, generating a response, and logging the message, response and time taken for computation. Then it uses the log reader function to read in all the messages from the chat history and creates an empty list where the formatted chat history will be stored. It then iterates through the original list, for either 9 items or the length of the list, whatever one is smaller. If the index it is on is odd, meaning that the message is from a user, the program will create a dictionary with the ‘type’ as ‘user’, the text as the message text and the id as a string denoting the number of the message. If the index was even, the dictionary is the same but with the type ‘ai’ instead. This dictionary is then appended to the formatted chat history list, which should now have 4 dictionaries in it due to the four messages in the chat history. The formatted chat history is then sent to the Jinja template ‘message.html’, where it is stored in the variable message.

Inside the messages container div, the Jinja script loops through the list and displays the correct message bubble for each message, with the correct text.

Extending the implementation to the rest of the program.

However, I suspect there is a problem with the current implementation, and that it has not been extended throughout the program. Below is the code for switching the theme of the page. This code, when it renders the message page, does not send any data over for the message variable, and because of this, even if there were messages being displayed, when the theme is switched the messages disappear. They are still there in the chat history, and next time a message is entered all available messages will show up, but changing the theme will clear the page of messages, which it shouldn't do.

```
43 @app.route('/theme', methods=['GET', 'POST'])
44 def theme_switcher() -> flask.Response:
45     """Switches the theme of the web app"""
46
47     global current_page, current_theme, stylesheet
48
49     if current_theme == 'light':
50         current_theme = 'dark'
51         stylesheet = 'static/dark-styles.css'
52     else:
53         current_theme = 'light'
54         stylesheet = 'static/light-styles.css'
55     return render_template(current_page, stylesheet=stylesheet)
```

To fix this, I have turned the formatted chat list into a global variable by declaring it at the start of the file, outside of any function, and then importing it into the message and theme switcher functions using the global argument. I then replaced the return statement in the theme switcher function with the following code that checks the page that is going to be rendered and adds in the messages variable using the updated formatted chat history if the page to be rendered is the messages page.

```
56     if current_page == 'message.html':
57         return render_template(current_page, stylesheet=stylesheet, messages=formatted_chat_history)
58     else:
59         return render_template(current_page, stylesheet=stylesheet)
```

TEST – theme switcher with messages

Here I am going to test the changes made above. To do this, I want to submit a few messages to the bot, getting a response each time, and then switch the theme, and ensure that the messages still appear, and that the page doesn't just turn blank.

Input – I will enter the message 'Hi, how are you doing today?', wait for the bot response, and then input another message, along the lines of 'I am doing great thanks.'. Once the bot has sent a reply to that, I will click the theme switcher button in the bottom left of the page to change the theme to dark mode.

Expected output – Once the theme switcher button is clicked, the page should change theme from the default light mode to dark mode, but the messages that were being shown before the button was clicked should keep being shown.

Actual output –



Figure 50

The program outputted responses to the two messages I entered into the message bar, and then once the theme switcher button was clicked, the page reloaded, this time with the dark mode colour scheme, and the message bubbles were still showing.

Here, the theme switcher function first checks with the global current theme variable which theme is currently being used, and swaps to the other theme, changing the stylesheet at the same time. It then checked if the page was the messages page, and as it was, it used the render_template method to render the current page (the messages page) with the new stylesheet and sent the formatted chat history global variable as the messages Jinja variable.

Search feature

As planned before, I will now add a search feature. This will help users to navigate through their previous messages, if there is one in particular, they want to find. This feature is particularly important as the message page can only show the last nine messages at once, and does not have any scrolling, so without a search feature, there would be no way for users to see any messages from any further back than 9 messages ago.

Adding extended logging

A problem, though, is that currently, only the last 20 messages are logged, and this is for context training purposes. So, with this approach, the program would only be able to search through the past 20 messages, which whilst still being an improvement on only being able to see the past 9, is still not enough – users should be able to view all past messages they have sent in the current session. There are other advantages to storing messages from further back as well, as it gives additional data that can be monitored for errors and used to help fix bugs. This is because it can help pinpoint where the error came from – whether the user input didn't make sense or was a type of sequence I know the model will struggle with, or perhaps the input sequence was perfectly fine, but the model outputted a completely unrelated sequence.

To add this, I need to change how the logging limiter functionality works. Currently the logging function will check if the log file is beyond the limit – 10 message and response pairs, and if it is, it will delete the file and recreate it. There are two problems with this approach. First of all, it will not work with the search feature as it deletes the old messages. But also, whilst emptying the log file once the file reaches the maximum row count achieves the original goal of preventing the file from ballooning in size as the conversation gets longer and longer, it means that once the tenth message is entered by the user, the previous messages will be cleared, meaning that no messages can be shown on the messages page. For the user, this would just seem like the chat history clearing at random points, which would appear to be odd behaviour and would mean the user would be able to see the message he had just replied to.

To fix these problems, two changes must be made. First of all, when the maximum file size is reached, only the oldest message should be removed, making the process of limiting the log file seamless, and not noticed by the user. Then, alongside messages being logged to the original log file, the messages should also be logged to a separate log file. Whilst this would mean that there would be a larger log file that would increase in size as the conversation went on and the bot was used more and more, it is a trade-off worth making as it can be very useful for logging, and the search feature would not work without it.

Whilst currently I could just use the same log file for both purposes, as the initial log file is used to provide context to the model, it should only contain the current conversation, otherwise the model would be attempting to understand a new input with unrelated context and would probably output nonsense. As it is difficult to find when the user closes the page in Flask, and easier way to do this is to log the date and time the AI's message was generated, and then if this was more than a small amount of time ago, it is most likely an old conversation.

```

102 def log(user_message: str, bot_response: str, time_taken: float) -> None:
103     """Logs the user's message, the bot response and the computation time to a CSV file"""
104
105     # Checking if either Log files do not exist
106     file_address = 'log.csv'
107     extended_file_address = 'ext_log.csv'
108     for file in [file_address, extended_file_address]:
109         if not path.exists(file):
110             create_log_file(file)
111
112     # Checking if the main Log file is full
113     with open(file_address, 'r', encoding='utf-8', newline='') as file_object:
114         limit_reached = sum(1 for _ in file_object) == 11
115     if limit_reached:
116         print('Log file limit reached')
117
118     # Reading in all rows from original file
119     rows = []
120     with open(file_address, 'r', encoding='utf-8', newline='') as log_file:
121         csv_reader = reader(log_file)
122         rows.extend(iter(csv_reader))
123
124     # Removing oldest message row and header row
125     rows.pop(0)
126     rows.pop(0)
127     remove(file_address)
128     create_log_file(file_address)
129
130     # Writing older rows to new Log file
131     with open(file_address, 'a', encoding='utf-8', newline='') as file_object:
132         csv_writer = writer(file_object)
133         for row in rows:
134             csv_writer.writerow(row)
135
136     # Logging new data to both log files
137     print('Logging data')
138     log_time = str(datetime.now())
139     log_file_writer(user_message, bot_response, time_taken, file_address, log_time)
140     log_file_writer(user_message, bot_response, time_taken, extended_file_address, log_time)
141
142 def log_file_writer(user_message: str, bot_response: str, time_taken: str, file_address: str, log_time: str) -> None:
143     """Logs data to any log file"""
144
145     with open(file=file_address, mode='a', encoding='utf-8', newline='') as file_object:
146         csv_writer = writer(file_object)
147         csv_writer.writerow([user_message, bot_response, time_taken, log_time])
148
149 def create_log_file(file_address: str) -> None:
150     """Creates the log file, adding in the header row"""
151
152     print('Creating new log file')
153     with open(file_address, 'w', encoding='utf-8', newline='') as file_object:
154         csv_writer = writer(file_object)
155         csv_writer.writerow(['User message', 'Bot response', 'Time taken', 'Log time'])

```

This is the new log function, which should be able to log new messages, responses, the computation time, and the log time. It should also be able to check if there are over 10 rows of messages and responses stored there, and if so, it will delete the oldest one. It will also create either of the log files if they don't currently exist.

Firstly, the function checks if either of the files do not currently exist, and if so, it uses the log file creator function written and tested earlier to create the file. The only change to this function was the addition of a log time header, which will store the exact time the message is being logged. The function then checks if the file is too long. If the file is too long, the function will first read in all the rows from the log file, by opening it in read mode, declaring an instance of the CSV reader class, and adding each row read from the file into a list called rows. It then removes the first two items from the list, which should be the oldest message and the header row. The function then deletes the CSV file and creates a new one using the CSV creator function, which works how it originally did, except with the addition of a log time column. The function then rewrites the rows of data from the original file (minus the oldest one of course) to the new file. Finally, the function writes the new data to be

logged to both log files, recording the date and time the data is being logged to before calling the same log writer function to write the data to both logs.

Whilst I did not have to have the log writing in a different function, it is an easy way to reduce the code that needs to be written, reducing file size and development time as well as how long it takes the computer to run the file, as the same code is used for logging for each file, the only difference being the file being logged to. Another option I had was to get the date and time of logging when the data was actually being logged, inside the logging function or even in the writerow method, as this would give a more exact time of when the data is being logged. However, the downside of this is that the time would be different for each file, despite the same data being logged at roughly the same time. Therefore, I get the date and time before calling each function and using that same date and time when logging to each file, in order to ensure consistency across log files which would make it easier to match data between files, in the event of the messages being the same.

TEST – logging data to both log files when the main file is non-existent.

Here I am going to test one of the scenarios in which the function would be used – logging when the main log file has not been created yet, which would be the case for the first message. This should produce the same results as the function did before the changes, however the data should be logged to the extended log file to, and the log time should be saved alongside the messages.

Input – I will input the user message ‘This is a test message’, and the bot response ‘This is a test response’, and the time taken of ‘25.4’ as arguments for the logging function, as this is quicker than opening the web app and logging a response generated by the model, and that is not behaviour which is being tested. The log.csv file has also been deleted.

Expected output – A file called log.csv should be created, with a header row including the headers ‘User message’, ‘Bot response’, ‘Time taken’, ‘Log time’, and the data inputted into the function stored in the row below beneath the appropriate column headers, along with a correct date and time for the log time column. Another file called ext_log.csv should also be created, with the same headings and the same data (including the same log time) in the row below.

Actual output – The log.csv file was created correctly, as shown below in Figure 51.

```
1 User message,Bot response,Time taken,Log time
2 This is a test message,This is a test response,25.4,2023-02-26 20:20:14.120477
```

Figure 51

As the screenshot shows, the user message, bot response and generation time have all been stored correctly, alongside a correct date and time. These have also been logged below the correct headings. The same has happened with the extended log file, with the file being created the correct headings being written and the data being written beneath, including the same log time as the other log file.

```
1 User message,Bot response,Time taken,Log time
2 This is a test message,This is a test response,25.4,2023-02-26 20:20:14.120477
```

Figure 52

TEST – logging data to both log files when main file is full.

While this was a feature of the older version of the logging function, the behaviour now works in a very different way, with the data from the old version of the log file being stored prior to the file's deletion, and all of the data being rewritten into the new file, apart from the oldest message and response pair, therefore leaving space for the new line of data to be written.

Input – Firstly, I have added another 9 messages to the log files, as shown below in Figure 53.

```
1 User message,Bot response,Time taken,Log time
2 This is a test message,This is a test response,25.4,2023-02-26 20:20:14.120477
3 This is another test message,This is another test response,25.8,2023-02-26 20:23:31.913639
4 This is a third test message,This is a third test response,25.8,2023-02-26 20:30:25.429674
5 This is a fourth test message,This is a fourth test response,25.8,2023-02-26 20:30:53.192868
6 This is a fifth test message,This is a fifth test response,25.8,2023-02-26 20:31:00.315156
7 This is a sixth test message,This is a sixth test response,25.8,2023-02-26 20:31:06.442857
8 This is a seventh test message,This is a seventh test response,25.8,2023-02-26 20:31:45.785457
9 This is a eighth test message,This is a eighth test response,25.8,2023-02-26 20:31:56.304537
10 This is a ninth test message,This is a ninth test response,25.8,2023-02-26 20:32:03.650712
11 This is a Tenth test message,This is a Tenth test response,25.8,2023-02-26 20:55:59.013712
```

Figure 53

Therefore, both log files have 10 messages in them, the maximum allowed. I will then input the user message 'This is a eleventh test message', 'This is a eleventh test response', and the time taken '18.9' as arguments for the log function.

Expected output – The extended log file should function as normal, with the data being written to the bottom of the file, just as all the other lines of data were. However, as the main log file is at its maximum capacity, the top line, including the user message 'This is a test message' should be removed, and the new line of data should be written into the bottom of the file, with the file therefore remaining the same length.

Actual output – The function works as intended, with the extended log file just adding an extra line for the new data, whilst the main log file deletes the first row of data before adding the new line on the end, as seen in Figure 54.

```
1 User message,Bot response,Time taken,Log time
2 This is another test message,This is another test response,25.8,2023-02-26 20:23:31.913639
3 This is a third test message,This is a third test response,25.8,2023-02-26 20:30:25.429674
4 This is a fourth test message,This is a fourth test response,25.8,2023-02-26 20:30:53.192868
5 This is a fifth test message,This is a fifth test response,25.8,2023-02-26 20:31:00.315156
6 This is a sixth test message,This is a sixth test response,25.8,2023-02-26 20:31:06.442857
7 This is a seventh test message,This is a seventh test response,25.8,2023-02-26 20:31:45.785457
8 This is a eighth test message,This is a eighth test response,25.8,2023-02-26 20:31:56.304537
9 This is a ninth test message,This is a ninth test response,25.8,2023-02-26 20:32:03.650712
10 This is a Tenth test message,This is a Tenth test response,25.8,2023-02-26 20:55:59.013712
11 This is a eleventh test message,This is a eleventh test response,25.8,2023-02-26 21:01:15.149770
```

Figure 54

Backend search algorithm

Now I need to actually search through the extended logs.

This will be written inside a separate function in app.py, called search(). As this function is inside the @app.route decorator that listens for the '/search' URL endpoint, the function will be called when the HTML form with the '/search' action is interacted with. A preliminary version of the function is shown below.

```
64  @app.route('/search', methods=['GET', 'POST'])
65  def search() -> flask.Response:
66      """Renders searched message plus number of surrounding messages"""
67
68      global messages_to_display
69
70      query = request.form['search-bar']
71
72      history = log_reader('ext_log.csv')
73      location = history.index(query)
74      messages_to_display = [history[i] for i in range((location - 4), (location + 4))]
75
76      current_page = 'search_result.html'
77      return render_template('search_result.html', stylesheet=stylesheet, messages=messages_to_display)
```

As before with the formatted chat history, the messages to display variable, which will store the messages around the searched message that will be displayed as a result of the search input, has been made a global variable. This means that it can be used (with the same contents) inside the theme switcher function. If I had not done this, as originally happened with the chat history, changing the theme would remove the chat history from the page, which shouldn't happen as the theme switcher function should change the theme and do nothing else.

The message being searched for is then requested from the HTML input element with the id 'search-bar'. The function then reads the extended log file into the history variable, using the log reader function from app_functions.py. This is the same function that was written before, except it has been modified to be able to read any log file. This has been done by simply changing the file address used in the log reader function to an argument that can be inputted into the function when it is called. So, inside the message generator, in order to get the context, the function is called with the argument 'log.csv' as only the main log file is needed. However, in this function, the extended log file is needed, so the address for that file is inputted into the function.

The function then finds the location of the searched message in the history list, and creates a list of messages to display, which consists of the four messages before the searched message, the searched message, and the four message after. This shows the user the message they searched for, as well as the context around the message.

Finally, the function renders a new HTML page called 'search_history.html' with the current stylesheet and the messages to display being sent across as Jinja variables.

TEST – Selecting and displaying correct messages.

Here I will test the above function. For testing purposes, the search_result HTML page is simply a copy of the messages page. Whilst this is only temporary, it is an already made page with the purpose of displaying a list of nine items in separate bubbles, which is the same purpose of the search result page, although that will be in a slightly different format.

This test is to simply check if the above function can correctly choose the 9 messages to be displayed and send them to the HTML template.

Input – I will first add extra messages to the log files using the log function created and tested before, in order to better represent the scenario of the function searching through a large log file of messages. The result of this is Figure 55, which shows the current contents of the extended log file. I will then input one of the messages in the log file – ‘This is a Tenth test message’ into the search bar on the index page.

```
1 User message,Bot response,Time taken,Log time
2 This is a test message,This is a test response,25.4,2023-02-26 20:20:14.120477
3 This is another test message,This is another test response,25.8,2023-02-26 20:23:31.913639
4 This is a third test message,This is a third test response,25.8,2023-02-26 20:30:25.429674
5 This is a fourth test message,This is a fourth test response,25.8,2023-02-26 20:30:53.192868
6 This is a fifth test message,This is a fifth test response,25.8,2023-02-26 20:31:00.315156
7 This is a sixth test message,This is a sixth test response,25.8,2023-02-26 20:31:06.442857
8 This is a seventh test message,This is a seventh test response,25.8,2023-02-26 20:31:45.785457
9 This is a eighth test message,This is a eighth test response,25.8,2023-02-26 20:31:56.304537
10 This is a ninth test message,This is a ninth test response,25.8,2023-02-26 20:32:03.650712
11 This is a Tenth test message,This is a Tenth test response,25.8,2023-02-26 20:55:59.013712
12 This is a eleventh test message,This is a eleventh test response,25.8,2023-02-26 21:01:15.149770
13 This is a twelfth test message,This is a twelfth test response,18.6,2023-02-27 15:21:57.411440
14 This is a thirteenth test message,This is a thirteenth test response,18.6,2023-02-27 15:22:21.622964
15 This is a fourteenth test message,This is a fourteenth test response,18.6,2023-02-27 15:22:34.406018
```

Figure 55

Expected output – A new page should be loaded under the URL endpoint ‘/search’, which should appear the same as the messages page, except with different messages being shown in the bubbles. The nine messages in the bubbles, from top to bottom, should be ‘This is a eighth test message’, ‘This is a eighth test response’, ‘This is a ninth test message’, ‘This is a ninth test response’, ‘This is a Tenth test message’, ‘This is a Tenth test response’, ‘This is a eleventh test message’, ‘This is a eleventh test response’, ‘This is a twelfth test message’.

Actual output -

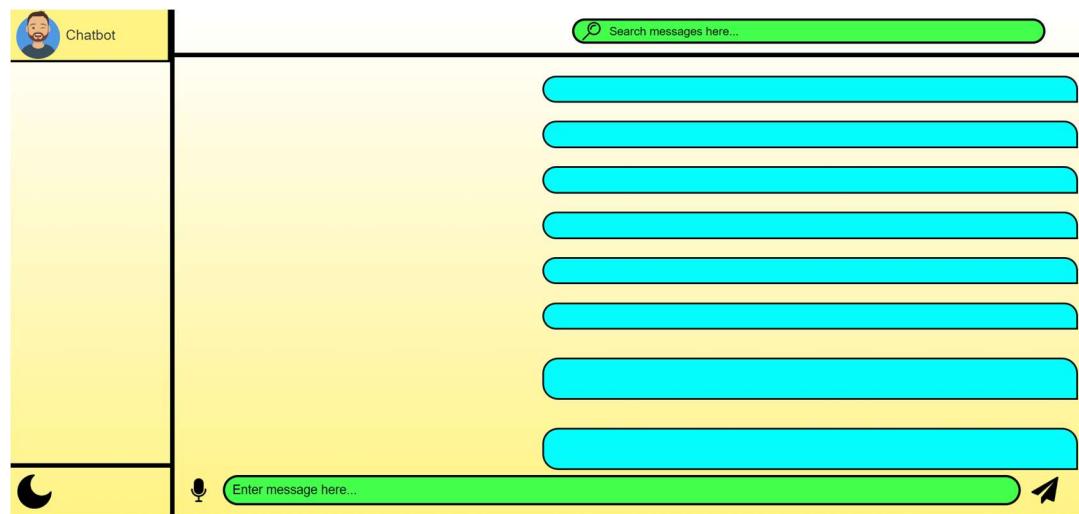


Figure 56

The function has not worked correctly, as the HTML page expects a very different input – a dictionary which specifies whether a message is from the AI or the user.

Iteration 2

To fix this and ensure a correctly formatted version of the chat history is used, I have changed the log reader function, as seen below.

```
9  def log_reader(file_address: str, format: bool=False, len_limit: bool=False) -> list:
10     """Reads the contents of the log file into a single list, ignoring the time taken and log time columns.\n11     Also has the option of formatting the chat history, turning each message into a dictionary,\n12     specifying if the message came from the model or the user"""
13
14     # Checks if the chat history is empty
15     if not path.exists(file_address):
16         print('No log file found')
17         return None
18
19     print('Log file found')
20     with open(file_address, 'r') as log_file:
21         csv_reader = reader(log_file)
22         chat_history = []
23         for row in csv_reader:
24             # Ignores header row
25             if 'User message' in row:
26                 continue
27             # Appends the first two entries in the row to the chat_history list
28             chat_history.extend((row[0], row[1]))
29
30         chat_history.reverse()
31
32     print(chat_history)
33
34     if format:
35         formatted_chat_history = []
36         if len_limit:
37             for i in range(min(9, len(chat_history))):
38                 if i % 2 == 0:
39                     formatted_chat_history.append({'type': 'ai', 'text': chat_history[i]})
40                 else:
41                     formatted_chat_history.append({'type': 'user', 'text': chat_history[i]})
42         else:
43             for i in range(len(chat_history)):
44                 if i % 2 == 0:
45                     formatted_chat_history.append({'type': 'ai', 'text': chat_history[i]})
46                 else:
47                     formatted_chat_history.append({'type': 'user', 'text': chat_history[i]})
48
49     return formatted_chat_history
50
51     return chat_history
52
53 def message_id_generator(chat_history: list) -> list:
54     """Adds IDs to each message based on their location in the list"""
55
56     for i in range(len(chat_history)):
57         chat_history[i]['id'] = f'message-{i + 1}'
58
59     return chat_history
```

I have added two extra arguments – format and length limit – to the function. Format specifies whether the part of the code calling the log reader function wants the chat history to be returned already formatted. This uses the formatting that was originally only used in the message function in app.py, meaning that the code can now apply the same formatting whenever needed, without needing to paste in the same formatting code or even use a separate function – instead the ‘format=True’ argument simply needs to be added to the log reader function call. The format argument is false by default, meaning that if the argument is not specified in the function call, the chat history will not be formatted, therefore meaning the function will work as it did before the formatting option was added. Whilst I didn’t need to add the option as to whether the chat history will be formatted, other parts of the program, such as the message generation, need an unformatted version of the chat history. Therefore, if there was no option as to whether the

function formatted the chat history or not, a new function would need to be created, for the purpose of outputting unformatted chat history.

The same reasoning applies to adding the length limit argument. This specifies whether the formatted list should have a maximum length or not. If this argument is true, the formatted chat history will have a maximum length of 9 elements, as that is all that can be shown on the page. This is the same as how the function worked previously, however this behaviour is needed for a very different use case to message searching. When displaying messages in conversation, only the most recent ones need to be shown, so looping through the messages and stopping after 9 works for that purpose. However, with message searching, extra processing needs to be done on the chat history list first, as shown in the code block further down the page, so the size should not be limited until after that processing is done.

The other change I have made is moving the message id generation into a separate function, which is shown below.

```
53  def message_id_generator(chat_history: list) -> list:
54      """Adds IDs to each message based on their location in the list"""
55
56      for i in range(len(chat_history)):
57          chat_history[i]['id'] = f'message-{i + 1}'
58
59
59  return chat_history
```

Previously, the message id would be added to the dictionary for each message when the dictionary was being created, which worked perfectly fine for displaying messages in conversation, as every message in the chat history list would be displayed, and it was already in the correct order. However here the correct messages have not yet been extracted from the chat history list. If this was done before the list was processed, there would be ids given to all items in the list, which could potentially be more than 20 or 30. However the HTML template can only take ids from 1 to 9, and so the others would just be ignored. This would likely mean that the incorrect messages would be shown, and so adding message ids should be done afterwards.

I have also changed the search function, allowing it to work with the formatted chat history, as is shown in the code below.

```
59  @app.route('/search', methods=['GET', 'POST'])
60  def search() -> flask.Response:
61      """Renders searched message plus number of surrounding messages"""
62
63      global messages_to_display
64
65      query = request.form['search-bar']
66
67      history = log_reader('ext_log.csv', format=True)
68      for message_info in history:
69          if message_info['text'] == query:
70              location = history.index(message_info)
71
72      messages_to_display = [history[i] for i in range(location - 4, location + 5)]
73      messages_to_display = message_id_generator(messages_to_display)
74
75      current_page = 'search_result.html'
76      return render_template('search_result.html', stylesheet=stylesheet, messages=messages_to_display)
77
78  if __name__ == '__main__':
79      app.run(debug=True, port=5000, host='0.0.0.0')
```

The algorithm will now loop through the newly formatted message history, checking if the contents of the text key in the message dictionary currently being looked at is the same as the query message inputted by the user. If so, it takes the location of that dictionary in the list as the location of the searched message, and then puts every item within 4 items of the searched message into a secondary list, before using the message id generator on this.

I now need to test this new functionality.

Input – same as before.

Expected output – same as before.

Actual output –

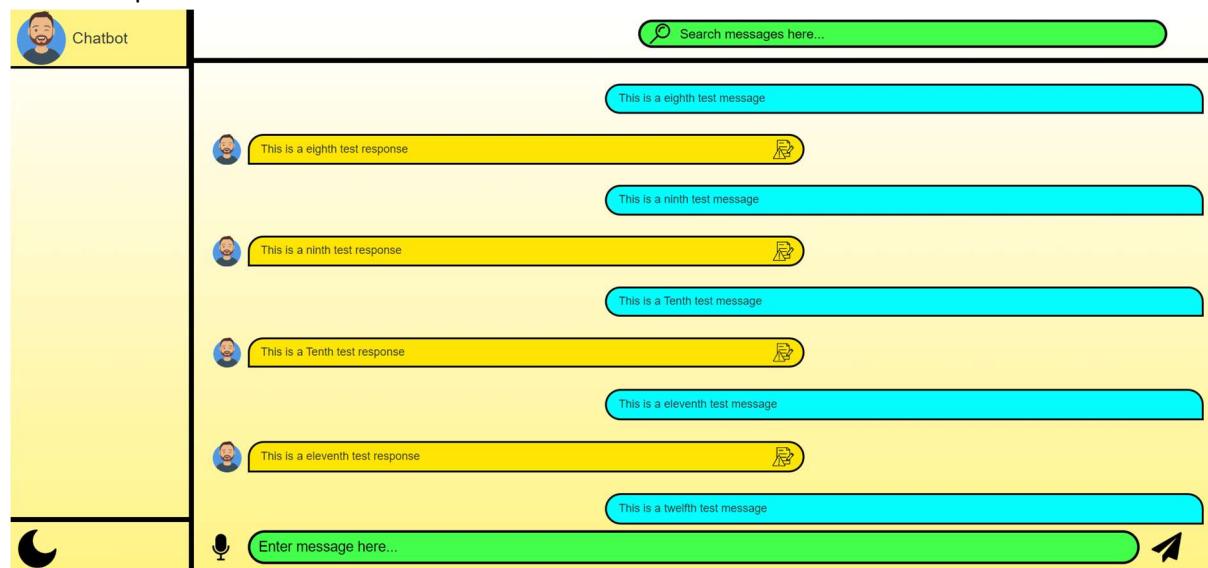


Figure 57

Here the algorithm has correctly displayed the searched for message, plus the previous 4 and next 4 messages. The code requested the message from the HTML input form for the search bar, and then used the log reader function to read into a variable the contents of the 'ext_log.csv' file, specifying that the messages should be formatted into dictionaries with a text key containing the text of the message, and a 'type' key containing either 'user' or 'ai'. The algorithm then loops through the list until it finds a dictionary where the value of the 'text' key is 'This is a Tenth test message'. When it finds that, the algorithm stores the location of that dictionary in the list into a separate variable, and then creates a new list which takes the items from the original list that use the ids between 4 before the searched for message and 4 after. Therefore, the new list should have nine items in it, all message dictionaries, with the searched for message exactly in the middle.

Iteration 3

Although this hasn't been tested for, another error that I know will be present in the algorithm is what happens when the user searches for a message that is either right near the start of the chat history, or right near the end. Following the algorithm above, if, for example, the query message was right towards the end of the chat history, then the incorrect messages would be displayed. The algorithm above ensures that the query message is always in the middle, with the 4 messages before and the 4 messages after being displayed. However, if there are less than 4 messages between the query message and the end of the chat history, then the algorithm would wrap round and display the first messages in the history once it has reached the end of the chat history. This is because location - 4, shown in line 72 of the python code above, would be a negative in this situation, and searching for a negative index of a list in python wraps round to the start of the list. The same would happen if one of the first messages in the chat history was searched for.

In order to fix this, I need a new algorithm to select the messages to be displayed, which is shown below.

```

192  def message_selector(length: int, location: int) -> int:
193      """selects the messages to be displayed when a message is searched for,
194      based off the length of the chat history"""
195
196      messages_before = 0
197      messages_after = 0
198      messages_after_query = length - (location + 1)
199      messages_before_query = length - (messages_after_query + 1)
200      if messages_after_query > 3 and messages_before_query > 3:
201          messages_after = 4
202          messages_before = 4
203      elif messages_after_query < 4 and messages_before_query > 3:
204          messages_after = messages_after_query
205          messages_before = 8 - messages_after
206      elif messages_after_query > 3 and messages_before_query < 4:
207          messages_before = messages_before_query
208          messages_after = 8 - messages_before
209
210      return messages_before, messages_after

```

First of all, two variables are created to hold the number of messages to be displayed before and after the query message – these variables should sum to 8, with the query message being the ninth

to be displayed. Then the number of messages in the chat history after the query message is calculated, which can then be used to calculate the number of messages before the query. If both of these are 4 or above, then 4 messages will be displayed before and after the query message. However, if there are less than 4 messages after the query message, then the number of messages to be displayed before the query message needs to be changed to reflect that, by taking the number of messages after away from 8, to ensure the total number of messages displayed is always the same. The reverse of this happens if there are less than 4 messages before the query, with the number of messages after being changed to ensure a consistent number of messages is displayed.

This function is located in the `app_functions` python file, and is called by `app.py` using the line below, which calls the function, inputting the length of the chat history list as the length argument, and the location of the message in the chat history list as the location argument.

```
messages_before, messages_after = message_selector(len(history), location)
```

I now need to test this to ensure the algorithm displays the correct messages.

Input – I will enter the string of text “Hi, what are you doing?” into the search box. This is the second most recent message, and as such there is only 1 message after it in the chat history, and not 4.

Expected output – The algorithm should return 1 for messages after and 7 for messages before.

Actual output – The algorithm correctly calculated that 1 message should be displayed after the query, and 7 should be displayed before.

Adding search result page

For testing purposes previously, the search result page has just been a copy of the message page, however this does not work for the user for a few reasons. The main problem is the fact that there is no way for the user to get back to the page showing the current point of the conversation without entering another message, as the message form has the '/action' message which calls the message function in app.py, which renders the message page. However, the user may want to see what the last few messages were to send a reply, and so they should have the option to go back to viewing the current conversation at any point. It also makes no sense for the user to be able to send messages from the search result page, and they should have to go back to the index or message page to do that. Therefore, the message input form at the bottom of the page should be removed and replaced with a button allowing the user to go back to the messages page.

The search results page is shown in Figure 58.

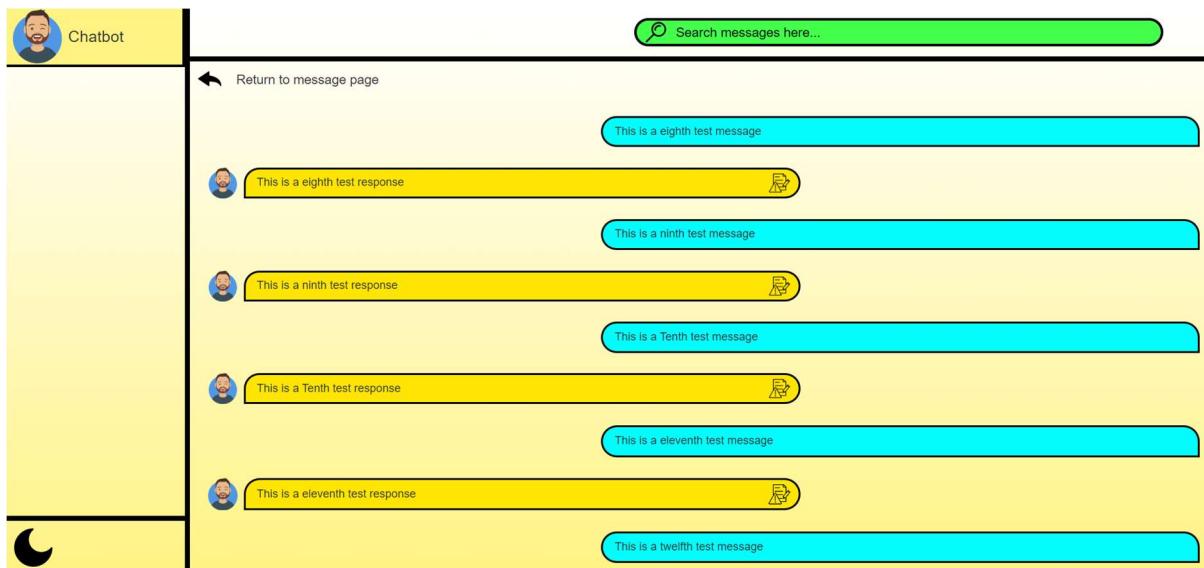


Figure 58

Here I have used the message HTML page as a template, keeping everything but the main section the same. I have used the same message container, with the same message bubbles, except here the grid the message container is in has changed, so that the message container is at the bottom, leaving space for the back button above the messages. Whilst using the same style message bubbles for the search results could be confusing as some users may think they are still on the messages page, and nothing has changed – their search hasn't worked – I believe that it is important to maintain consistent styling. This is important for a few reasons: obviously because it is more aesthetically pleasing and gives a sense of familiarity and branding, but also because the user is comfortable with the design and understands it already at this point. The user already knows that the blue bubbles are messages from them, and that the yellow bubbles are messages from the bot, and changing this could confuse the user.

Below is the HTML code for the main div in the search results HTML page (the rest of the code is the same as the other pages).

```
31      <div id="main-result">
32          <div id="back">
33              <form action="/back" id="back-form" method="post">
34                  <button type="submit" id="back-button">
35                      Return to message page
36                  </button>
37              </form>
38          </div>
39          <div id="messages-container-result">...
40          </div>
41      </div>
```

I have used a new id for the main div, 'main-result'. This is so I can use different CSS styling, as the CSS grid needs to be changed. As you can see from the CSS code below, the grid template has

```
display: grid;
grid-template-columns: 100%;
grid-template-rows: 5vh 85vh;
gap: 1vh;
```

it is the most prominent evidence to the user that they are not on the messages page, so showing it to them immediately, making it the first thing they read, should ensure they do not get confused.

The back div simply contains a form with the back button in it. Whilst in theory the back form should not have to be inside a div (in fact with the message submit form, the form is a grid item on its own and not inside any div), for some reason the styling of the messages would get messed up when I removed the back div, so until I work out why, it will stay as it is.

As you can see from the CSS, the back div occupies the top row of the CSS grid, and is a flex container itself. The use of a flex container allows me to more easily align the button inside, using line 81 to move it to the left of the div, and have it start from there instead of the centre. Line 82 just means that the contents of the div will be vertically centred.

The back form shown in line 33 of the HTML

code at the top of the page shows that the form has the action '/back'. This means that it will trigger the URL endpoint '/back', and therefore allows it to interact with a function in app.py.

changed, with a smaller, 5vh row above the larger, 85vh row. The smaller row will contain the back button, whilst the larger row will contain the message container. I believe that it is important to have the back button near the top of the page, as

```
76      #back {
77          grid-row: 1;
78          border: none;
79          text-align: center;
80          display: flex;
81          justify-content: left;
82          align-items: center;
```

Below is the CSS for the back button.

```
83  #back-button {  
84      border: none;  
85      background-image: url(light-images/back-icon.png);  
86      background-position: 10px center;  
87      background-repeat: no-repeat;  
88      background-size: contain;  
89      background-color: transparent;  
90      padding: 4px;  
91      padding-left: 50px;  
92      height: 3vh;  
93      width: 13vw;  
94      font-size: 2.3vh;  
95      box-sizing: content-box;  
96      margin-top: 1vh;  
97      white-space: nowrap;  
98  }
```

I have borrowed from the message search text box for this, as it fits the same general idea of a form element made up of an image on the left of some text. To do this, I added a background image and vertically centred it, but specified that horizontally, it should be 10px from the left, which should align it to the left of the button, and specifying no repeats and for the size to be contained inside the dimensions of the button means that it will only appear at the left of the button, and will not repeat across the button or spread to large. I use the 'padding-left' argument to move the text of the button, moving it 50px from the left edge of the button, therefore meaning it won't overlap with the image on the left side. I specify the height and width of the button in lines 92 and 93, and then use line 97 to specify the wrapping behaviour of the button. If I left out line 96, when the page was resized, therefore making the button smaller, the text would wrap onto another line below. This is not useful, as not only does it not look as good, but it also means that some of the text will be outside of the button, and therefore unclickable, which I feel is very unintuitive for the user. Specifying that there is no wrapping means that the text will stay on the same line, and all be clickable.

The python code for the back button is very simple, and is shown below.

```
80  @app.route('/back', methods=['GET', 'POST'])  
81  def back() -> flask.Response:  
82      """Renders message page when back button on search results page clicked"""  
83  
84      global formatted_chat_history, current_page, stylesheet  
85  
86      current_page = 'message.html'  
87      return render_template(current_page, stylesheet=stylesheet, messages=formatted_chat_history)
```

The back function is inside a decorator that calls the function when the '/back' URL endpoint is detected. The function simply gets the global variables for stylesheet, current page and the formatted chat history, updates the current page variable, and then renders the message page with the correct stylesheet and most recent message history.

TEST – Back button

The functionality of the back button needs to be tested.

Input – First, the main log file should be deleted, and I will start a conversation with the bot. Once there are more than 9 messages in the conversation, I will search for one of the messages in the conversation. Once the search results page appears, I will click the back button.

Expected output – The bot should respond to the messages as expected, and then when a message is searched for, the search results page should be rendered, with the searched for message plus the 4 messages before and the 4 after. Then once the back button is pressed, the messages page should be rendered again, displaying the same messages as before.

Actual output – First of all, I used the message input box to have a conversation with the bot. The log of the conversation is shown below in Figure 59.

```
User message, Bot response, Time taken, Log time
2 "Hi, how are you doing today?", "I'm doing well, thank you. I hope you are as well. How are you?", 43.61, 2023-03-08 11:37:05.183466
3 I am a bit tired today, I'm sorry to hear that. I hope you feel better soon. What's going on?, 28.43, 2023-03-08 11:38:44.280780
4 "I got a good amount of sleep last night, but I still feel tired for some reason", I'm sorry to hear that. Have you been sleeping a lot lately? Maybe you need more sleep., 21.97, 2023-03-08 11:45:43.9
5 I don't usually get enough sleep but last night I got a good 7 hours, "Wow, that's a lot of sleep. How did you manage to get that much sleep?", 21.97, 2023-03-08 11:45:43.9
6 "I was very tired yesterday, so I got to bed early yesterday.", That's always a good thing. Did you get enough sleep? I'm always tired after a long day of work., 21.99, 2
7 I felt like I did get enough sleep, That's good. Sleep is very important. Did you get a good amount of sleep?, 9.45, 2023-03-08 11:48:17.537363
```

Figure 59

I then entered into the search input box the string 'I don't usually get enough sleep but last night I got a good 7 hours', which was one of the messages I inputted (and therefore a user message). The search results page was then correctly rendered, with the four messages before the searched message, the searched message, and the four messages after the searched message all correctly shown. Then once the back button was pressed, the app returned to the messages page, showing the conversation exactly how it was before, and allowing me to enter messages into the message input box in order to continue the conversation.

Adding separate colour for query message bubble

In order to highlight to the user which message is the one they searched for, I will change the colour of the message bubble containing the searched-for message to light orange in the light mode style, and green in the dark mode style. Adding this will make it easier for the user to realise that they are in fact on the correct search results page, and to draw their attention towards the message they searched for, which should allow the user to understand the purpose of the page and understand the surrounding messages and their context more quickly. Whilst I don't necessarily need to add this, as there is already a button that says, 'Return to message page', therefore telling the user that they are not on the message page, I still believe that adding a separate-coloured bubble for the query message should make the page easier to understand for the user.

I have done this using the python code below, which is in the search function of app.py.

```
94     for message in messages_to_display:
95         if message['text'] == query_message:
96             message['class'] = 'query-message-box'
97         else:
98             message['class'] = False
```

This loops through every message in the list of messages to be displayed on the search results page and checks if the text of the message is the same as the user's original query. If it is, a new key called 'class' is created, and its value is set to 'query-message-box'. On the other hand, if the message's text is not the same as the query message, the 'class' is simply set to false. Whilst the value stored in the 'class' key, when the message is not the query message, is not important as long as it does not relate to any of the other classes used, I felt that simply setting it to false is a good way to show that the message is not the query message, whilst not possibly overlapping with another class name, and ensuring that the key isn't empty or non-existent in order to not cause any errors.

I then change the classes of the message boxes to include the value of the class key in each message, as shown below in the example for the AI message boxes.

```
45     <div class="ai-message-box message-box {{ message.class }}">
```

I then use CSS to add a background colour to the new class 'query-message-box', making it light orange in the light stylesheet, and green in the dark stylesheet. For example, the code for the dark stylesheet is shown below. I have set the argument as important to ensure it overrides the default

```
160 .query-message-box {
161     background: #0fb500 !important;
162 }
```

background colour of the message bubble.

TEST – Separate colour for query messages

Here I need to test that the message bubbles change colour correctly, in accordance with whether the message is the query message or not.

Here, the algorithm should loop through each message in the list of messages to display and set the 'class' value to false for all of them, except for the query message, which should have the value 'query-message-box'. The Jinja variable in the search results HTML file should then detect if the class key of the message being displayed has the 'query-message-box' value, and if so, include it as one of

the classes of the message box div. The CSS should then detect that and set the background of the message box to the appropriate colour.

Input – First I will enter the string ‘What kind of programming do you want to do? There are so many different ways to do it.’, which is a message from the AI, which was sent previously, into the message search box. Once the search results page has been rendered, I will then click the theme changer button.

Expected output – When the string of text is entered into the search box, the search results page should render, with all the bubbles being correctly coloured (blue for user messages, and yellow for AI messages) except for the ‘What kind of programming do you want to do? There are so many different ways to do it.’ message, which should be light orange. Then once the theme switcher button is clicked, the page should refresh and load in dark mode, with all the bubbles being coloured as usual (blue for AI and red for user) apart from the bubble containing the string of text above, which should be green.

Actual output -

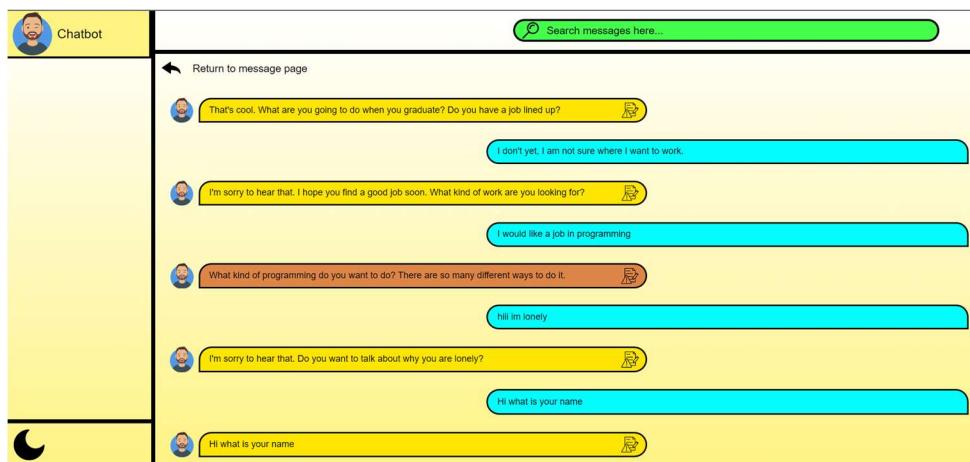


Figure 60

As you can see in Figure 60, the search results page has been correctly rendered, with the AI bubbles as yellow and the user bubbles as light blue, except for the query message, which has been rendered in orange. Figure 61 below shows the results of clicking the theme switcher button, with the page rendered in dark mode correctly, with the AI bubbles in dark blue, the user bubbles in red and the correct query message in green.



Figure 61

Adding check for if searched message exists

A necessary feature to prevent errors is for the program to check if the message the user searched for exists. This is important as if this was not done, and the user searched for a message that does not exist, the program would throw up an error as it would not be able to locate the searched-for message in the chat history list. This is something that must be avoided, particularly because the error is simply a technical message in the console, and the web app will just crash.

Inside the search function, after the section of code that finds the location of the searched-for message, I added the code shown below.

```
78     if not location:
79         current_page = 'search_error.html'
80         return render_template(current_page, stylesheet=stylesheet, message=query)
```

As the section above creates a variable called location only when it finds the query message in the list of messages, this means that if the query message is not found, the location variable will not exist. Therefore, in order to check if the query message was found, I can simply check if the location variable exists. If it does not, I change the global variable for the current page to 'search_error.html', the file name for the error page, and then render said error page, using the correct stylesheet and sending across the searched-for message, so it can be used in the error message to the user.

The main div of the error page is shown below.

```
31         <div id="main-result">
32             <div id="back">
33                 <form action="/back" id="back-form" method="post">
34                     <button type="submit" id="back-button">
35                         Return to message page
36                     </button>
37                 </form>
38             </div>
39             <div id="error-container">
40                 <div id="error-icon"></div>
41                 <p id="error-message">
42                     The message you searched for - '{{ message }}' - does not exist
43                 </p>
44             </div>
45         </div>
```

The HTML code is mostly the same as the search results page, the only difference is the contents of the main div. Whilst in the search results page, the main div was a grid containing two items – the back button and the displayed messages. However, in the error page, whilst the back button is still there of course, the displayed messages have been replaced with an error container. This container div contains two elements, a div for the error icon, and the error message encased in paragraph tags. The error icon div is empty, with the icon being added as a background image using the CSS shown below. The error message is simply string of text, containing the Jinja variable message, which will hold the message the user originally searched for.

I used the following CSS to style the error container and its contained elements.

```
75 #error-container {  
76     grid-row: 2;  
77     margin-left: 1vw;  
78     margin-right: 1vw;  
79     display: grid;  
80     grid-template-columns: 100%;  
81     grid-template-rows: 47.5% 47.5%;  
82     gap: 5%;  
83 }  
84 #error-icon {  
85     background-image: url(Light-images/error.png);  
86     background-position: center;  
87     background-repeat: no-repeat;  
88     background-size: contain;  
89     background-color: transparent;  
90     grid-row: 1;  
91 }  
92 #error-message {  
93     grid-row: 2;  
94     font-size: 2.5vh;  
95 }
```

The error container is positioned in the second slot of the main grid, just as the messages container was in the search results file. I then use the display attribute in line 79 to turn this element into a grid container itself, with 1 single column and two rows, each row taking up 47.5% of the available height. There is also a gap between the grid boxes of 5% of the available height. Whilst I did not have to add a gap, it is a simply way to add separation between grid elements and is simpler and more consistent than adding padding or margins in the elements themselves. I use the styling in lines 85 to 90 to create the icon, adding it as a background image, centring it in the container

div and ensuring it doesn't repeat or expand out. The error message itself is given a relative font size, so that when the page is resized, the text still stays prominent on the page, but also proportionate – not oversized or tiny depending on how the page was resized. This allows me to retain consistency even when the size of the page is manipulated.

I did not have to use a grid for the container, I could have just used a flexbox or even left the elements in there bare, however using a grid meant I could control their sizing and positioning better, especially in comparison to simply leaving the elements bare with no container. Whilst in theory using a flexbox could have worked, I found that as the div for the error icon did not have an absolute size, and could not inherit one from the flex container, that as it in theory had no content (only a changed background) the div would just default to having no size. Therefore, a grid was an easier way to position the elements whilst giving the icon div an absolute size, no matter the content, so that it would not just automatically shrink to nothing and be able to display its background image. As I have discussed before, I am using background images instead of HTML img tags so that the address of the image is stored in the CSS file, so changing the CSS stylesheet changes the image being used. This is the method I am using for changing the theme of the page when the user wants to.

TEST – search error page

Here I need to test if the error page works properly. First of all, I need to ensure that the correct page shows, and the algorithm doesn't just attempt to display the standard results page and throw up an error because it cannot find the query message. I also need to ensure that the error messages use the correct query message. And then finally I need to ensure the back button works as intended (although this shouldn't even need to be tested as it is the exact same as the search results page, using the same python function and HTML form).

Input – I will enter the string of text 'This is a test search' into the search box, which is not a message that appears in the extended log file.

Expected output – The error page should be rendered, displaying the error message 'The message you searched for – 'This is a test search' – does not exist.'

Actual output – when the ‘This is a test search’ string was inputted, the error page was rendered with the correct error message, as is shown below in Figure 62. Therefore, the algorithm works as intended.

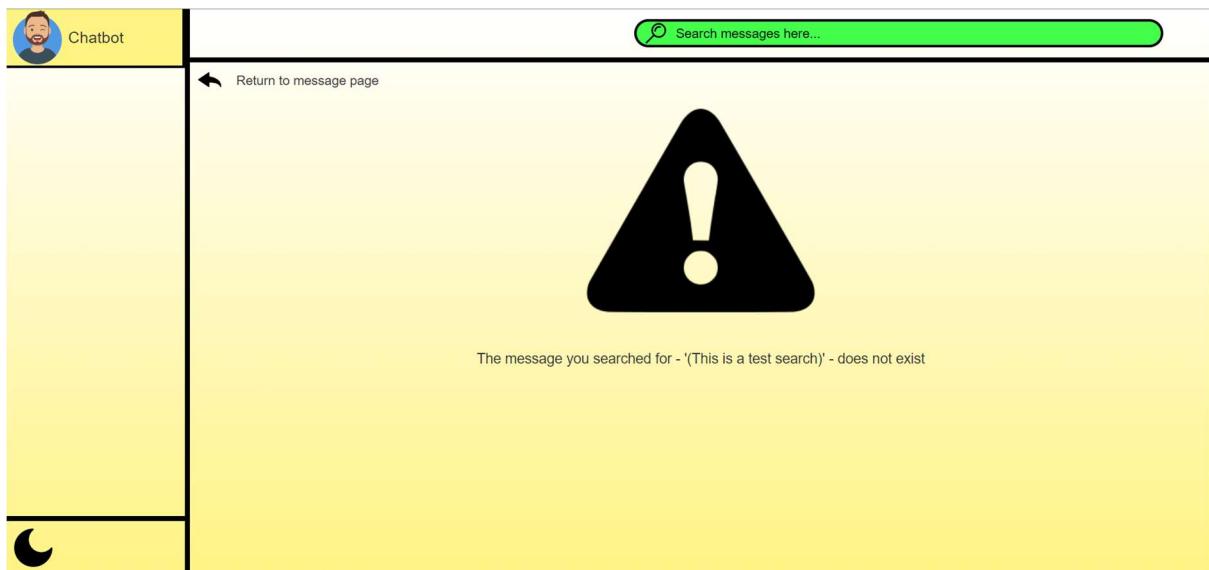


Figure 62

Adding search suggestions

One current problem with the search functionality is that it requires the exact message to be entered into the search box, including the same punctuation and spacing, meaning that if the message is entered, but with one word spelt slightly differently, or if there is no comma where there was in the actual message, the program will return an error saying that it cannot find the message searched for.

My method of fixing this is by using a HTML Datalist, which is a HTML element which creates a list of possible responses under a text box, and the list of possible responses changes as the user types into the input box, only showing relevant responses.

There are other possible methods of dealing with this, such as using Levenshtein distance.

Levenshtein distance is an algorithm that can calculate the separation between two strings of text (the minimum number of single character edits needed to change text into another). This may fix the problem as even if the user enters a search term which isn't exactly the same as the message they were searching for, however there are downsides. For one, there is always the possibility that Levenshtein distance may find a closer message than the one the user intended. This is why using a Datalist may be a better approach, as it gives the users options. It lets them see all possible messages to find the one they are attempting to search for, ensuring they always find the one they are intending to search for. Furthermore, implementing a Datalist is also simpler than Levenshtein distance. Levenshtein distance requires implementing the algorithm in the Python script, and implementing that in the search algorithm, which takes up more development time and has more room for errors. On the other hand, a Datalist is simply a HTML element that contains all the possible list items, making it much easier and quicker to implement, especially with the help of Jinja.

To implement the Datalist, I use the following function written in app_functions.py.

```
234     def get_messages_list() -> list:
235         """Returns a list of individual messages"""
236
237         history = log_reader('ext_log.csv', format=True)
238         history.reverse()
239
240         return [message['text'] for message in history]
```

This function uses the log reader function discussed earlier to get a formatted list of all messages in the extended log file, which it then reverses. It then returns a list of the value of the text key for every message dictionary in the formatted list, so there is a list purely containing every message sent either by the bot or the user, in separate list items.

Then the return statement shown below from the main function (which loads the initial page when the app is first opened), shows that the function is called when a page is rendered, and the output of the function is transferred to the Jinja variable 'messages_list'.

```
136     return render_template('index.html', stylesheet=stylesheet, messages_list=get_messages_list())
```

The HTML code for adding in the Datalist is shown below.

```
24      <form action="/search" method="post" autocomplete="off" id="search-form">
25          <input type="text" id="search-bar" name="search-bar" placeholder="Search messages here..." list="messages-list">
26          <datalist id="messages-list">
27              {% for message in messages_list %}
28                  <option value="{{ message }}">
29              {% endfor %}
30          </datalist>
31      </form>
```

This code shows the new HTML form for the search bar, now with the Datalist included. The list is referred to in the input element (line 25), and then created in lines 26-30, where, inside a Datalist element, a Jinja for loop is used to create an option element for every item in the messages list.

TEST – Datalist

Here I need to test if the Datalist appears when the search box is typed into. As the Datalist is an element with direct support in Google Chrome, the list should carry over the Chrome design theme.

Input – I will open the webapp, and on the index page, click on the search bar.

Expected output – A list of messages from the extended log file should appear when the search box is clicked on. As the first few messages in the extended log file are ‘This is a test message’, ‘This is a test response’, ‘This is another test message’, ‘This is another test response’, and so on, these are the messages which should appear at the top of the list.

Actual output -



Figure 63

The list shown on the left-hand side of Figure 63 continues down and contains all messages and responses from the extended log file. The list then adapts when I start entering text, as shown below in Figure 64.

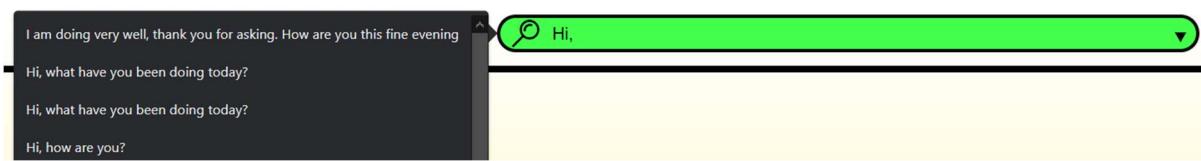


Figure 64

Therefore, the Datalist works as intended, giving an adapting list of suggestions from the extended log file.

Adding message reporting

Message reporting is an important feature for my program, as it gives user's a way to give feedback on how the program runs, and report errors, which can make bug fixing and further development easier. It also lightens the development load, as much of the continuous testing that would normally need to be done after the program is published can, in effect, be done by the users instead.

When the message and search result HTML files were created, there were already report forms added in, which consisted of a text box for users to enter a report reason (this would popup in a Bootstrap modal). However, the problem posed here is identifying the message that the user is reporting, as the only data the python program can get directly from the HTML form is the report reason, and it has no way of knowing the message the user intended to report.

To fix this, I have changed the name of the input element (the text box) to depend on the id of the message the form is in. I have done this using Jinja code, as shown in the HTML file shown below.

```
57      <form action="/report" method="post" autocomplete="off" class="report-form">
58          <input type="text" class="report-reason-input" name="report-{{ message.id }}" placeholder="Enter report reason here...">
59      </form>
```

As the name has changed, and not the class, this does not interfere with any CSS, only changing what Flask sees the form as being called. This means that from the name of the form that the user has submitted into and that has posted data to the Flask app, I can work out the message that the user is attempting to report.

To directly get the id of the message from the name of the form, I use the following python code, which is in the report function in app.py (inside the @app.route decorator which listens for the '/report' URL endpoint, which is what the HTML form triggers).

```
139     reported_message = list(request.form.to_dict().keys())[0]
140     report_reason = request.form[reported_message]
141     report_message = reported_message.replace('report-', '')
```

The Flask function 'request.form.to_dict' gets the name of all forms triggered, and their data, and converts it into a dictionary, which I can then use the .keys() method on to turn into a list of all the dictionary keys (therefore a list of all form names). As there is only one form triggered, I get the first value from the list of keys, which should be the name of the report form for the message the user intended to report. Then, in line 140, I use that name to get the data from the form specified, before removing the 'report-' substring from the start of the form name, so that the string just becomes the message id.

As there is already the dictionary of messages being displayed (with the message text, the origin of the message, and the id of the message) that is stored as a global variable, allowing it to be accessed from within the return function, whilst still being the exact same as when it was sent to the messages page to display messages. This means that using the python code below, I can search

```
143     for message in chat_history:
144         if message['id'] == report_message:
145             location = chat_history.index(message)
146             bot_response = message['text']
147             user_message = chat_history[location - 1]['text']
```

through the list of message dictionaries to find the one that contains the correct id and store the text of this message as the bot response variable (the origin of the message doesn't need to be

checked as reported messages will always be from the bot). I also store the location of the dictionary containing the reported message in a separate variable, so I can use that to find the text of the message before, which will be the user's message that triggered the reported message from the bot. This should be stored for error checking purposes so that I have a better idea of what went wrong.

I then use the two functions below (written in app_functions.py) to store the reported data into a CSV file.

```

196  def log_report(user_message: str, bot_response: str, report_reason: str):
197      """Saves reported response, user's message and report reason to a CSV file"""
198
199      if not path.exists('report.csv'):
200          create_report_file()
201
202      with open('report.csv', 'a', encoding='utf-8', newline='') as file_object:
203          csv_writer = writer(file_object)
204          csv_writer.writerow([user_message, bot_response, report_reason])
205
206  def create_report_file() -> None:
207      """Creates report file, adding in the header row"""
208
209      print('creating report file')
210      with open('report.csv', 'w', encoding='utf-8', newline='') as file_object:
211          csv_writer = writer(file_object)
212          csv_writer.writerow(['User message', 'Bot response', 'Report reason'])

```

The log report function will be called from the report function in app.py, with the appropriate user message, bot response and report reason inputted as function parameters. Firstly, the function checks if the file exists, and if not, a separate function called create_report_file will be used to create the file. This function opens the file in write mode, which means that if the file cannot be found, a new one will be created with the name specified. An instance of the CSV writer class is then created, which is used to write the header row to the newly created, currently blank, CSV file.

Once the main function is sure the file is created, it opens it in append mode (meaning that any data written will be added on to the end of the file) and creates an instance of the CSV writer object which is used to write the new row of data. This row consists of the user message, bot response and report reason, all separated by commas.

The final aspect of the report process is the report page. As all Flask functions must render a page as the function's return argument, a new page should be created, which will thank the user for submitting a report and contain a back button allowing the user to easily return to the messages page. Whilst a separate report page isn't strictly necessary, and the report function could just link back to the messages page for example, having a separate report page is useful. It gives additional feedback to the user that their report was processed, which is further confirmed by the page directly stating the message that the user reported. If this page wasn't there and the program just routed straight back to the messages page, the user may be unsure if their report was processed, as they haven't received any feedback that it has.

The report page is mostly the same as the search error page, with the main div consisting of a back button, and a separate div which contains an image and a message. The HTML code for the main div is shown below.

```

33     <div id="main-result">
34         <div id="back">
35             <form action="/back" id="back-form" method="post">
36                 <button type="submit" id="back-button">
37                     Return to message page
38                 </button>
39             </form>
40         </div>
41         <div id="report-container">
42             <div id="report-icon"></div>
43             <p id="report-message">
44                 Thank you for reporting the following message - '{{ message }}'
45             </p>
46         </div>
47     </div>

```

As you can see, the back button is identical, but the difference is in the div below. This div contains two elements, an icon, and a message below it. The message makes use of a Jinja variable to refer explicitly to the message the user reported, whilst the icon is simply an empty div, with the icon set as a background image using the CSS below.

```

93 #report-icon {
94     background-image: url(light-images/post-report.png);
95     background-position: center;
96     background-repeat: no-repeat;
97     background-size: contain;
98     background-color: transparent;
99     grid-row: 1;
100 }

```

TEST – reporting AI message.

Here I need to test whether the above algorithm, spanning HTML pages, CSS files, and multiple Python files, works as intended. This means that when the user clicks on the report icon and enters a message into the text box in the modal that pops up, this should then trigger the return function in app.py. This function should then get the name of the input element the user used and use that to find the id of the message being reported. Using this id, the algorithm should get the text of the message being reported, as well as the text of the user's message that triggered the reported response, and use the logging functions to save the message, reported response and report reason to the report CSV file (and create the file with the correct headings if it does not already exist). Finally, the report page should load, which should directly refer to the reported message whilst thanking the user for their feedback.

Input – I will enter the message ‘Hi, how is your day going?’ to the bot, once it responds, I will click on the report icon for the bot’s message and enter the string ‘Test reason’ into the text box for entering a report reason that will be inside the pop-up box. Then if the report page loads, I will click the back button which should appear at the top of the page.

Expected output – when I enter a reason into the text box and hit enter, three things should happen. First of all, as the report CSV file does not already exist, the program should create one with the appropriate header row (User message, bot response, report reason). The program should then

write the user message 'Hi, how is your day going?', the bot's response to that message, and the report reason 'Test reason' to the CSV file in the line below the header row. Finally, the report page should be loaded, with text on it that should directly contain the message reported. When the back button is clicked on that page, the program should return to the message page, which should still only have two messages on it – 'Hi, how is your day going?', and the bot response.

Actual output –

Figure 65 shows the bot's response to the input message.

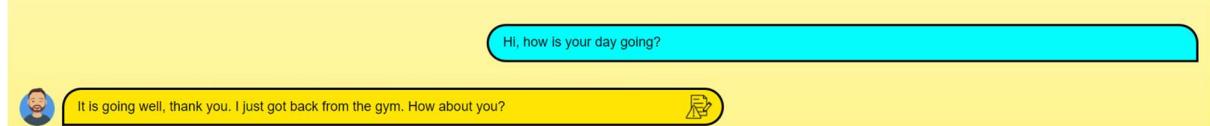


Figure 65

Figure 66 shows the contents of the 'report.csv' file.

```
1 User message,Bot response,Report reason
2 "Hi, how is your day going?", "It is going well, thank you. I just got back from the gym. How about you?", Test reason
```

Figure 66

Figure 67 shows the report page which is rendered after the report reason is submitted.

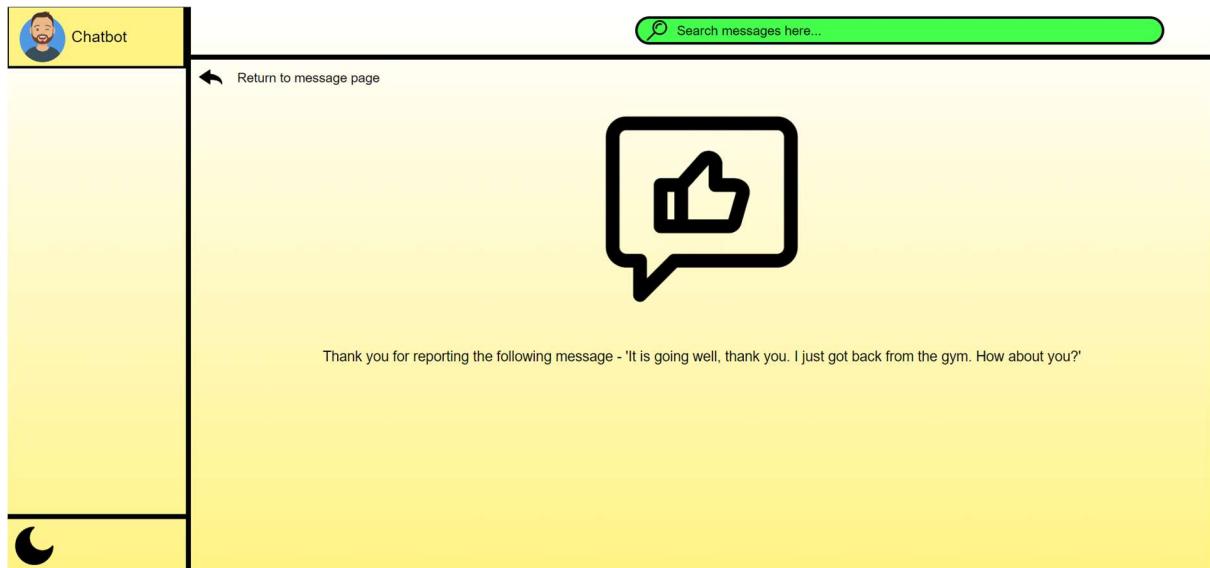


Figure 67

Figure 68 Shows the message page as it was shown when the back button was clicked .



Figure 68

As shown by Figure 66, the user's message, the correct bot response and the report reason have been correctly stored in a newly created CSV file, and Figure 67 shows that the report page has been rendered correctly. Therefore, the algorithm works as intended.

Adding grammar checker

Another feature I am going to add to my chatbot is a grammar checker. This would be a simpler algorithm that would fix spaces around full stops, and capitalise the first letter of every sentence, just like how automatic grammar fixing on mobile phone keyboards or apps like WhatsApp. This serves two purposes: firstly, it corrects mistakes the user may have made in order to create a better, more easily understandable message. But more importantly, it makes the message easier for the bot to understand and therefore output an equally understandable response. It can also make the messages easier to search for, as even with the Datalist, an exact match needs to be entered into the search bar. Therefore, if the message is already using proper grammar, it will be easier for the user to search for, as the user's search will most likely not consider any possible grammar errors.

As this algorithm will also be applied to the bot's responses, it can be used to correct any possible grammar mistakes before the user sees them, making the bot seem smarter.

One error the bot frequently makes is unnecessary whitespace, especially at the start of a response. This is particularly important as it is incredibly unlikely the user will notice this error, and as such they will never be able to search directly for the message. The unnecessary white space can occur either before the start of the text, or around punctuation marks. To fix this, I use the code below which uses regex to correct any whitespace around punctuation marks, so that there is no space before and only space afterwards. I then use the strip method to remove whitespace from the start and end of the string.

```
132     message = re.sub(r'\s(?=[\.,:;])', "", message)
133     message = message.strip()
```

However, adding capitalisation at the start of each sentence is a trickier task. The simple way to do it is to simply use the split string method to split the string at full stops, creating a list of sentences, before capitalising the first letter in each list item and then combining the list back together into one string. But this does not work in all cases. Firstly, this would need to be done for all forms of punctuation that can end a sentence, like an exclamation mark or question mark. But also, not every full stop ends a sentence, for example the full stop at the end of 'MR.' isn't the end of the sentence. In order to account for these edge cases, I will use the NLTK (natural language toolkit) library, which has a sentence tokenizer that can accurately split strings into lists of sentences. Whilst using NLTK has its downsides, the most notable of which being that it will increase the size the program will take up on the user's computer as it is a large extra library which will need to be downloaded, it is the most accurate method of splitting strings of text into sentences, which can then be capitalised and joined back together, as I have done in the Python code below.

```
134     sent_tokenizer = nltk.data.load('tokenizers/punkt/english.pickle')
135     sentences = sent_tokenizer.tokenize(message)
136     sentences = [sent.capitalize() for sent in sentences]
137     message = ' '.join(sentences)
```

First, I have loaded NLTK's sentence tokenizer, before using it to split the string into a list of sentences. I have then created a new list, which consists of a capitalised version of every sentence in the list, which is then joined back together.

These algorithms have been combined together to create one function, which is written in app.py.

```
129 def format_message(message: str) -> str:
130     """Formats message to capitalise and remove whitespace and fix some grammar errors"""
131
132     message = re.sub(r'\s(?=[\.,:;])', "", message)
133     message = message.strip()
134     sent_tokenizer = nltk.data.load('tokenizers/punkt/english.pickle')
135     sentences = sent_tokenizer.tokenize(message)
136     sentences = [sent.capitalize() for sent in sentences]
137     message = ' '.join(sentences)
138
139     return message
```

This function will be called on the user input as soon as it is requested from the HTML form, and on the bot's response before the message generator function returns a response.

TEST – grammar checker

Here I need to test whether the grammar checker works properly. To do this I will enter a message to the bot which will contain grammatical errors and see the bot's response. I will then add the grammar checker back in and see the bot's response with both the user's and bot's messages being auto fixed.

Input – The message ' hi . what have you been doing today?' will be inputted into the message input bar twice, once with the gramma fixer and once without it.

Expected output – Without the grammar checker, the user input should appear in the message bubble exactly how it was typed, and the bot should output an appropriate message, that may or may not contain grammatical errors. I will then add the grammar checker in, and the when the user string is entered, 'Hi. What have you been doing today?' should appear in the user message bubble and the bot should output an appropriate response with no grammar errors.

Actual output – The output without the Gramma fixer is shown below in Figure 69.



Figure 69

The text inside the user's message bubble is exactly the same as what was entered with the grammatical errors still there, and there is an additional space before the bot's response.

The output with the grammar fixer is shown below in Figure 70.

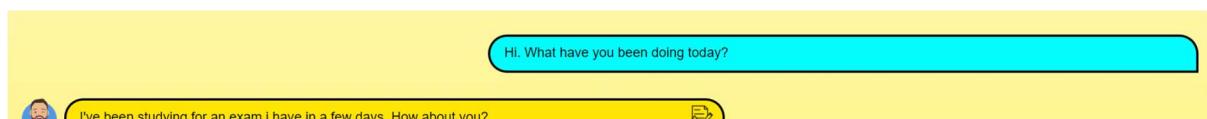


Figure 70

The user's response has had its grammar corrected, and there is not extra space before the bot's response, showing that the grammar checker has worked correctly.

Speech-to-text

The final large feature to add is speech to text. This gives user's another way to communicate with the model, allowing hands-free communication to make interacting with the model as easy as possible.

This will be split into two algorithms: recording the message and converting it to text.

Recording the message will be done with the Sound Device and Sound File packages. These are useful for two reasons: they are quick to use, only taking up a few lines of code to record and save a message and allow you to specify exact parameters like sampling rate. This is useful because the model I will use later for converting the recorded sound file to text was trained with a certain sample rate, and so the input into that model should match that sample rate. Therefore, using Sound Device and Sound file means that I can ensure the audio will match the necessary parameters for the model to be able to process it correctly.

Audio recording

The audio recording will be a separate process to the conversion to text and will simply consist of audio being recorded from the microphone for 5 seconds, before being saved as an audio file which can then be processed by the text to speech model. I have chosen 5 seconds because I believe it is enough time for the user to say any message they may want to communicate. Whilst I could make it longer to accommodate for any longer messages the user may want to send, this would lead to the user being sat there awkwardly in silence for a while waiting for the recording to stop, which would obviously not lead to an enjoyable experience.

The function in app_functions.py to record a message is shown below.

```
245 def record_message() -> None:
246     """Records all audio from microphone for 5 seconds"""
247
248     sample_rate = 16000
249     duration = 5
250     file_name = r'temp_audio.wav'
251     frames = sample_rate * duration
252
253     print('recording')
254     my_data = sd.rec(frames, samplerate=sample_rate, channels=2)
255     sd.wait()
256     print('recording ended')
257     sf.write(file_name, my_data, sample_rate)
```

Firstly, the sample rate is set to 16000, or 16KHz, the duration is set to 5 seconds and a file name is set to a set location. The point of doing that is to ensure that as it is saved to the same file name in the same location each time and will therefore be overwritten in each time (in the edge case that for some reason the file isn't automatically deleted like it should be). Whilst this isn't technically necessary for the program to work, it prevents files from being endlessly created, which clogs up the program folder, making it more complex and look more daunting and harder to comprehend to the user or anyone trying to understand or use the program, whilst also increasing the file size.

I then specify the number of frames that the program should record, by multiplying the sample rate by the duration. Next, I use Sound Device's record method to record audio. I set the time the audio

should be recorded for using the frames variable and input the sample rate. Then Sound Device's wait method means that the program pauses until the recording is finished. Finally, I use Sound File's write method to save the recorded audio, using the specified file path and sample rate.

TEST – Recording audio.

Here I need to test that the function created above can record audio, and then save the audio file correctly. To do this, I will call the function, which should automatically start recording audio. I will then say a test statement into the microphone, and after five seconds the program should stop recording. The program should then create a file called 'temp_audio.wav' in the program directory, which should contain an audio recording with me saying the test statement.

Input – I will call the function, and then say the phrase 'This is a test statement'.

Expected output – A file called 'temp_audio.wav' should be created in the program directory containing a recording of me saying 'This is a test statement'.

Actual output – An audio file was automatically created after the function finished running, called 'temp_audio.wav', in the correct directory. This contained a 5 second audio clip of me saying 'This is a test statement'.

Therefore, the function can record and save audio correctly, thus fulfilling its intended purpose and allowing it to record user messages in order to be transcribed by the speech recognition function.

Automatic speech recognition

I will then use another transformer model to convert the audio file into a string of text that can be inputted into the message generation functions. This will be a different transformer model, this one trained for automatic speech recognition. As mentioned above in the design section, the Wav2Vec model will be used, and for now it will not be fine-tuned. First, I will implement and test a pre-trained, non-fine-tuned version, and if that is not good enough, I can fine tune it.

Below is the code for the automatic speech recognition function in `app_funcitons.py`.

```
266 def asr_transcribe() -> str:
267     """Uses pre-trained Wav2Vec transformer for automatic speech recognition on given audio file"""
268
269     file_name = r'temp_audio.wav'
270     model_name = 'facebook/wav2vec2-base-960h'
271     model = Wav2Vec2ForCTC.from_pretrained(model_name, cache_dir='data/models')
272     processor = Wav2Vec2Processor.from_pretrained(model_name, cache_dir='data/processors')
273
274     speech, sample_rate = load_data(file_name)
275     print('Loaded speech file')
276     # Tokenize
277     input_values = processor(speech, return_tensors="pt", sampling_rate=sample_rate).input_values
278     print('Processed input values')
279     # Take Logits
280     logits = model(input_values).logits
281     print('Processed logits')
282     # Take argmax
283     predicted_ids = torch.argmax(logits, dim=-1)
284     # Get the words from predicted word ids
285     transcription = processor.decode(predicted_ids[0])
286     print('Decoded transcription')
287     # Correcting the letter casing
288     transcription = format_message(transcription)
289
290     print(f'Transcription: {transcription}')
291
292     remove('temp_audio.wav')
293
294     return transcription
```

First the file name (name of the file created by the recording function that would be called first) and name of the model to be used are declared, and the model and processor functions are imported from the Transformers library (not shown in screenshot as they are imported at the top of the file).

Then the model and processors are declared, using the Wav2Vec transformer, and more specifically a version trained using connectionist temporal classification (CTC) as the model. Connectionist temporal classification is used for training recurrent neural networks where the timing of the sequence being trained is variable. This is therefore a useful technique as the dataset being used for training doesn't have to be adjusted, and the neural network is more used to dealing with variable length sequences. The processor acts as a wrapper that combines a feature extractor and a tokenizer together, giving the functionality of both. So therefore, using the processor isn't necessary as I could just use the tokenizer and feature extractor separately, however the processor means that less methods need to be imported, and the tokenizer and feature extractor do not need to be downloaded separately.

The model and processor are declared in much the same way as the model and tokenizer for BlenderBot, however there has been one change to the declaration process, with a cache directory

```
119     model = BlenderbotForConditionalGeneration.from_pretrained(name, cache_dir='data/models')
```

being specified. This is where the model and tokenizer should be stored when they are downloaded for the first time, and where the program should check for them before it starts to download them again. This is useful for two reasons: first of all, it means that all data for the program (including downloaded models) is kept in the program folder instead of being stored in other parts of the user's hard drive. This is useful for the user as if they want to be able to see how much storage the program is taking up, or delete the program, it is easier as only one folder needs to be deleted and then all data from the program will have been deleted. The other reason this is useful is that it prevents the program from downloading the models and tokenizers again each time a message is entered in the case it cannot find the model data. This was an issue I found coming up for both transformers, and so I have added this argument for when both transformers are used, as you can see from the line of code below which is taken from the model generation function.

I then use the `load_data` function shown below to load the audio file.

```

259  def load_data(input_file: str) -> np.ndarray:
260      """Loads speech data from audio file"""
261
262      speech, sample_rate = sf.read(input_file)
263      return speech, sample_rate

```

This function uses the `read` method from the Sound File library to load both the audio data and the sample rate of the sound file inputted into the function, and returns them.

I then use the Wav2Vec processor to process the speech and get the input values for the model. To do this, I input the audio data from the audio file (the `speech` variable), and the sample rate of the audio file into the processor. Finally, I specify that the processor should return the input values in the form of PyTorch tensors, as just like with the text generation with BlenderBot, I am using PyTorch throughout the program. The processor will then return the input tensors that can be inputted into the model. The processor will return a one item dictionary, with the key '`input_values`' and the value being a set of PyTorch tensors that make up the input values. So therefore I specify that the `input_values` variable should contain the value of the '`input_values`' key in the outputted dictionary.

I then pass the tensors into the model. Just like with the processor, the model outputs a dictionary. An example of which is shown below in Figure 71. The model outputs a lot of additional information

```

CausalLMOutput(loss=None, logits=tensor([[[ 11.6247, -24.6794, -24.4534, ..., -6.4065, -6.3
030, -7.0640],
[ 11.1118, -24.3973, -24.1928, ..., -6.6127, -6.2532, -7.0293],
[ 11.2146, -23.7819, -23.7258, ..., -6.2554, -4.2679, -6.8589],
...,
[ 10.1333, -22.7914, -22.5762, ..., -7.1022, -5.3787, -5.9308],
[ 10.1021, -22.0458, -21.8340, ..., -6.5495, -5.4095, -6.4137],
[ 10.2852, -22.0491, -21.7967, ..., -6.2577, -5.3462, -6.4926]]], grad_fn=<ViewBackward0>), hidden_states=None, attentions=None)

```

Figure 71

about the input, including the loss parameter, any attention parameters and hidden states. However the actual processed result from the model is stored in the `logits` key, so

I specify that the `logits` variable should contain the data from that key of the output.

Next I use PyTorch's `argmax` function to find the maximum value of the `logits`, and therefore the most likely prediction.

Finally, I decode the chosen prediction into human readable text, using the processor that was originally used to encode the input.

I also call the `format_message` function to format the transcription to ensure that it is more easily understandable by the message generation model. At the end of the function, once the transcription has been completely generated, the audio file is no longer needed, so the `remove` method from the OS library is used to delete it. This prevents the directory from being clogged up with unnecessary audio files, and therefore stops the file size of the program from increasing as much.

TEST – converting audio to text.

Here I need to test that the Automatic speech recognition function can correctly transcribe user's audio messages into text, allowing the response generation algorithm to create a statement in response to the user's input. I will test this feature using not only the audio recording function, but also in conjunction with the response generation function, allowing me to see how both work together, and whether the transcriptions are good enough for the message response algorithm.

Whilst I could test this function on its own, it is more useful to test it in conjunction with the message response algorithm because I do not need the transcriptions to be perfect, I just require them to be understandable by the response algorithm, and so testing them in conjunction means that I can easily work out if the transcriptions are good enough.

I will test the entire feature using the web app, where I will open the app onto the index page, and then click the microphone button. Doing this should automatically call the message recording function to record and save the message for five seconds. Then the ASR function will be called, which will load in the audio file, run it through the Wav2Vec processor to get the input values, run it through the actual model and get the logits from the models output containing the predictions. The `argmax` function will then be used to find the most likely prediction, which will be decoded, with the function returning a string of text – the transcription of the message said by the user. Finally, this will be run through the response generation algorithm, and the transcription and response will be displayed in message bubbles.

Input – I will say the following statements to the program after clicking the microphone button – ‘Hi, how are you doing today?’, ‘What is your name?’, ‘What have you been up to this morning?’.

Expected output – I would expect the algorithm to create a good enough transcription that it is understandable by the user to be close to the original inputted sequence, and good enough for the response generation model to be able to output an intelligible response.

Actual output – Attempting to transcribe any sequence returns the following exception:
‘`RuntimeError: Expected 2D (unbatched) or 3D (batched) input to conv1d, but got input of size: [1, 1, 80000, 2]`’.

As the error appears to be with the number of dimensions Wav2Vec is using or expecting to use for its convolutions, it is most probably an issue with the backend programming of either the Wav2Vec model provided by Hugging Face, or with the programming of the version of PyTorch I have been using. As this means that the error is most likely not with my implementation of Wav2Vec, there is little I can do to fix this algorithm, without using a completely different model. As such, for now this feature will have to be removed, and perhaps in the future, with either a working version of Wav2Vec, or a different model, this feature could be added back in.

Testing installation

The program will be stored on a public GitHub, allowing users to easily download the necessary files off there, however fully installing the program requires a few extra steps the user will have to do themselves.

All data required for the program to work are either stored in the GitHub repository, or will be downloaded by the program's files, apart from the actual python application and interpreter, which will need to be downloaded from the python website by the user separately beforehand. Using markdown, I created a guide to installing the program, which appears on the GitHub page as shown below in Figure 72.

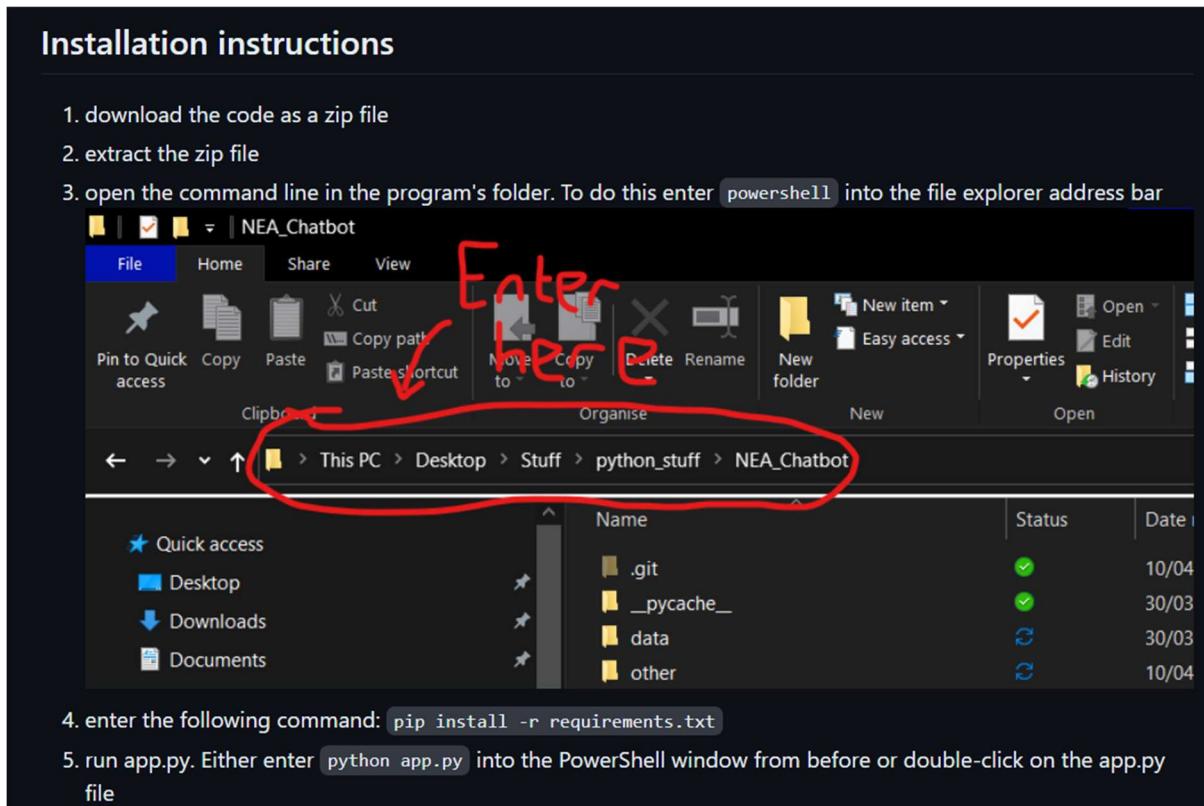


Figure 72

First the GitHub repository will be downloaded into a zip file. Whilst the repository could contain all of the program's files, I have chosen to not store some of the files, such as the 'data' folder, which contains the BlenderBot model and tokenizer, as well as the log files. This is because they can be automatically generated by the program whilst it is running, and therefore there is no point in storing them in the GitHub repository, and if anything, it is a bad idea, as adding unnecessary to the repository means the user will have to download more of the program in one go, thus increasing installation time. Therefore, these files have been left out and the program will have to download them itself.

I then opened a PowerShell window in the program's folder and entered the command in the readme, as you can see below in Figure 73.



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

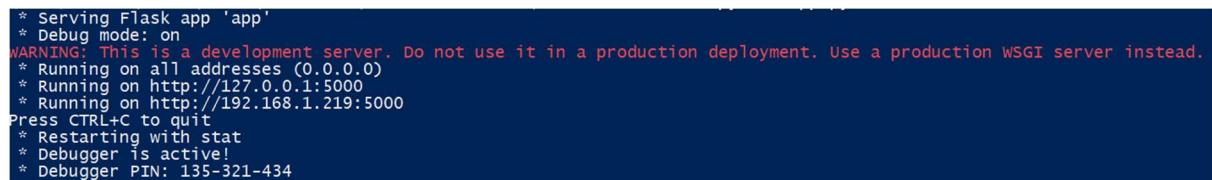
Try the new cross-platform PowerShell https://aka.ms/pscore6
PS C:\Users\kazik\Downloads\NEA_Chatbot-main\NEA_Chatbot-main> pip install -r requirements.txt
```

Figure 73

This command uses the requirements.txt file that I created when developing the program to install all the required libraries and their dependencies. It can be created using the command 'pip freeze > requirements.txt', which creates a text file in the program folder listing all the packages in the development environment, preventing the issues of missing packages and dependencies.

All packages installed correctly and now the program can be run for the first time.

To run the program, the user should either enter 'python app.py' into the currently open PowerShell window, or simply double click on it. When running it, the following paragraph pops up in the PowerShell window.



```
* Serving Flask app 'app'
* Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:5000
* Running on http://192.168.1.219:5000
Press CTRL+C to quit
* Restarting with stat
* Debugger is active!
* Debugger PIN: 135-321-434
```

Figure 74

The user should enter either of the addresses (either 'http://127.0.0.1:5000' or 'http://192.168.1.219:5000') into the address bar of their browser, and the program's home page renders, with both log files being created at the same time. When the user enters a message into the chatbot for the first time, the required machine learning models are downloaded, and the response is generated. Whilst this does mean the first message will take longer to generate, all other messages are now much quicker.

As this has worked as intended, the program has shown that it can work correctly on a new installation, and therefore users should be able to use it after downloading it from the GitHub page, without having any errors or extra steps to deal with.

Post-development testing

I will now run through the entire program, testing all features. This will be based not only on the functionality of the program, whether the features work as intended, and fixing any bugs, but also on the usability of the program. Can users easily understand how to use the program, and can they unintentionally break it with unexpected inputs, that any user could do. To do this, I will also test for robustness – checking whether putting in unexpected inputs, such as messages that don't exist into the search bar, can break the program, or at least create confusing responses for the user.

When I first open the program, the index page appears as expected, apart from a spelling error in the user information present in the middle of the page, which is shown in Figure 75.

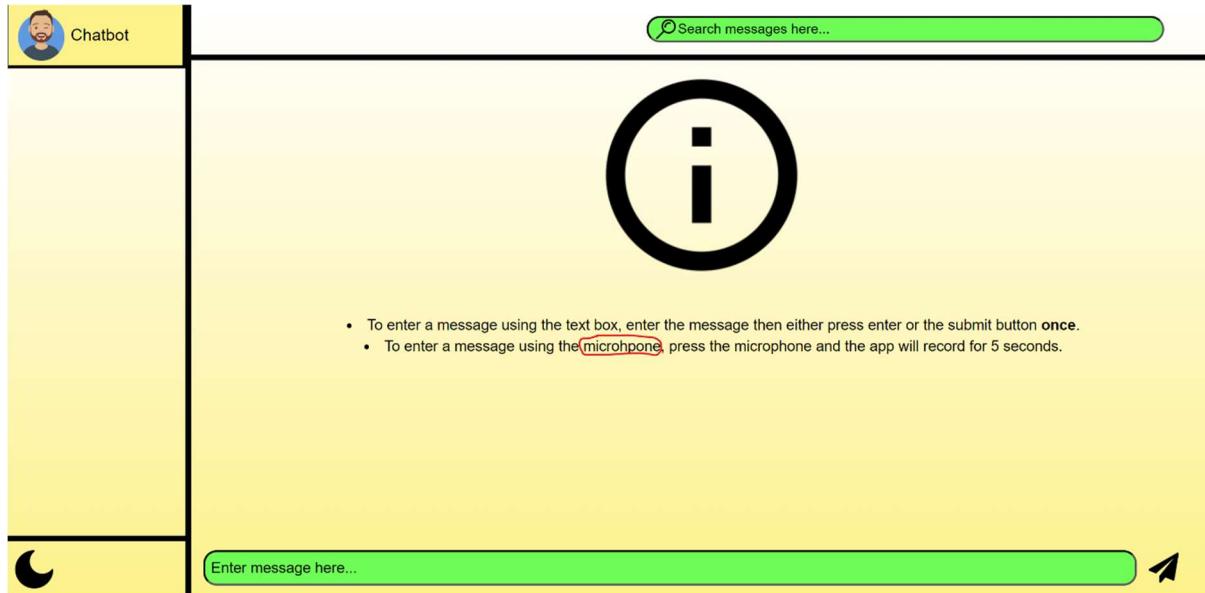


Figure 75

This has been corrected, changing 'microhpone' to 'microphone'. I also tweaked the padding in the search bar, adding extra padding to the left to give extra separation between the placeholder text and the search icon in the search bar. I also added some extra padding on the left and right of the message input bar, to give extra space between either the placeholder text or any long message entered in, and the sides of the message bar. The index page now appears as shown in Figure 76, with the changes highlighted in green.

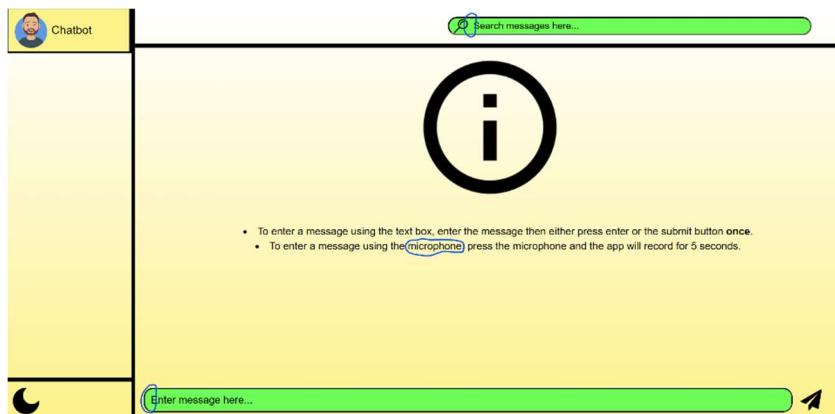


Figure 76

I then entered messages into the message bar.

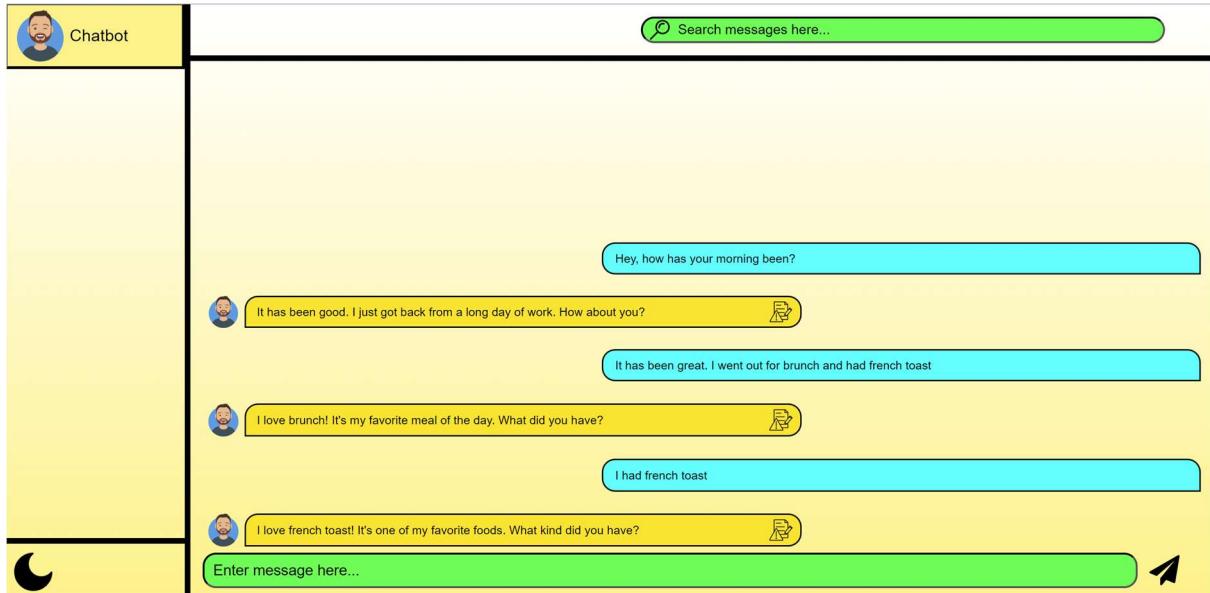


Figure 77

The responses were all easily understandable, and in good English. The model mostly understood the input sequence and generated an accurate response. In Figure 77, it made a slight slip up by not seeing that I specified what I had for brunch in the initial message about it, asking for clarification where it was not needed. However, the responses were within a good degree of accuracy that I am happy with. Also, as I used a more powerful computer, the responses were generated faster, all taking around 8-10 seconds, which shows that the program can work well on a reasonably powerful computer.

I then tried entering a very long message, which worked as intended. The text scrolled across in the text box when I was typing, and when the message and response were displayed, all the text fit nicely into the message bubble. The bot still understood the long message, and responded well to a part of it, answering a question that was at the start of the input sequence, which is the natural thing for it to do.

However, the bot did struggle when presented with obviously nonsense messages.

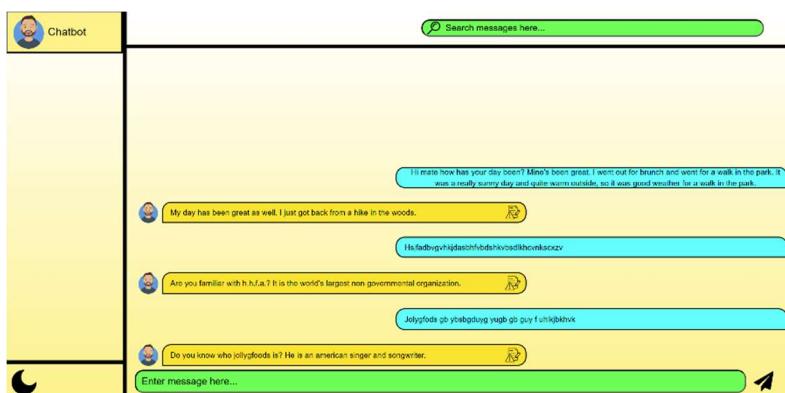


Figure 78

message is nonsense.

As you can see with the bottom two message and response pairs in Figure 78, the model still attempted to find meaning in certain parts of the nonsense message. Whilst on one hand the fact it can recognise certain names and other things even if the spelling is incorrect, as shown in the last message from the bot, the bot should still be able to understand that the

In order to fix this, I added a short new function to `app_functions.py`.

```
172 def english_check(message: str) -> bool:
173     """checks if the message is understandable english by tokenizing it into words, and checking if at least one of them is in the dictionary"""
174
175     # Used NLTK word tokenizer to split the message into a list of words
176     words_list = word_tokenize(message)
177     print(words.words())
178     return any(word.lower() in words.words() for word in words_list)
```

The English check function uses NLTK's word tokenizer to split the message into a list of words, and then iterates through the list of words, and the function returns true if any of the words are in the English dictionary. Whilst I could have just used the `split` method native to python to create the list of words, NLTK's word tokenizer is better as it also accounts for hyphenated words like 'up-to-date', which is not in the English dictionary together, but is when split up, and is still relatively correct English. Then in `app.py`, I use the following code to run the user-inputted message first, and only use the model to generate a response if the checker function returns true. On the other hand, if it returns false, the model just outputs a hardcoded 'That makes no sense' string and sets the model generation time to 0.

```
70     if english_check(user_message):
71         reply, time = reply_generator(user_message)
72     else:
73         reply = 'That makes no sense'
74         time = 0
```

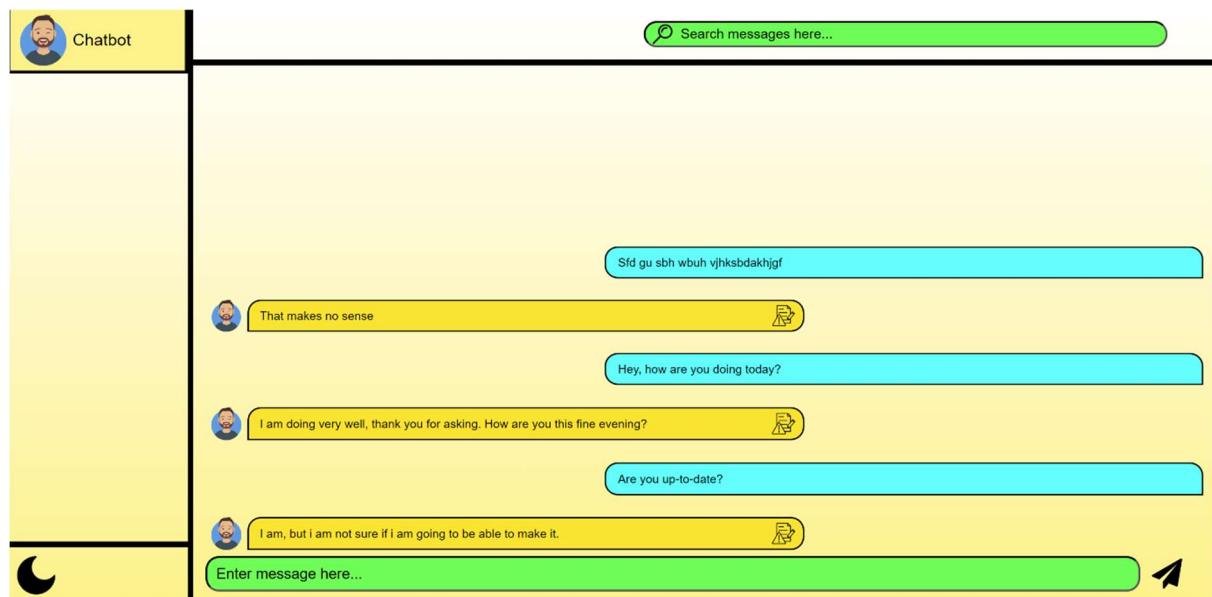


Figure 79

As shown in Figure 79, the function works correctly. The first message shows it correctly picking up a nonsense message and returning the correct string in response. The second message shows that it still lets correct messages be sent to the model for response generation, and the third message shows that the NLTK tokenizer has correctly split up 'up-to-date' into its constituent words. The issue is now fixed, and the program can correctly handle a nonsense input. This serves another purpose as well, as it covers the situation of the user entering nothing (just pressing submit without entering any text).

One problem with the message input, is that if the user submits their message twice, either by clicking the button multiple times, or some combination of the button and the enter key, the program will interpret them as separate messages, with the model attempting to generate a

response separately to each one. Not only does this mean multiple repeated messages will be displayed on the messages page, but also that as the model generation function (the most resource intensive part of the program) is being run multiple times simultaneously, it can cause the user's computer to slow down drastically, and possibly crash. However, as it is tricky to detect when the page is loading in flask, and flask only changes the page in the return argument of a function, this is difficult to fix. A warning has been added to the home page, and the user will have to heed that to prevent this error.

I then tested the search feature, and it performed correctly. The Datalist was accurate, showing all messages that had been sent using the program in this installation, and the results page was also accurate. It correctly identified the query message, and showed the messages around it, without wrapping round to the other end of the list of messages. The return to message page button also works, even when multiple searches have been made one after the other, without returning to the message page. I did however tweak the return button, changing it to say, 'Return to conversation', as shown in Figure 80 below. I also made this edit to the search error page, and the report page.

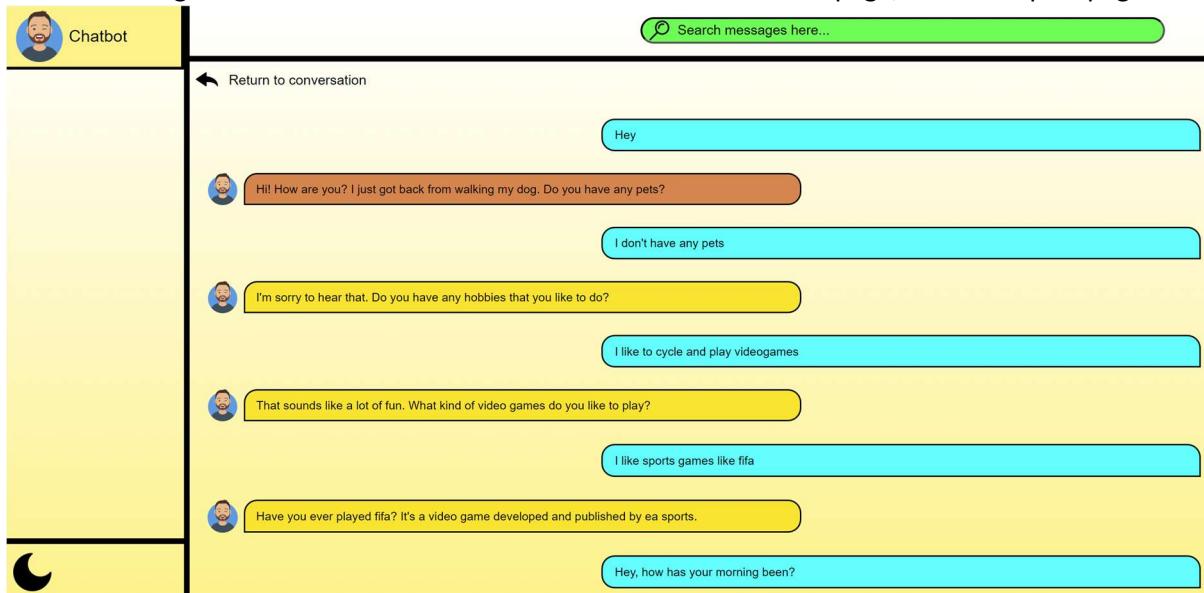


Figure 80

One issue with the search results page is that it cannot separate different conversations. This means that it may show the end of one conversation, and the start of another, even though the messages in the other conversation are completely irrelevant to the searched message. However, this is not really fixable, and is a more fundamental flaw in how messages are logged. I also don't believe this should matter too much, as users should be able to easily identify separate conversations and disregard those messages easily themselves.

The search function is also good at picking up errors and incorrect searches, with any string searched that doesn't exactly match a message in the log file causing the error page to display. The return to conversation button also works on that page.

The report feature works correctly, with the modal popping up when the button is pressed, and the report page being rendered when the report reason form is submitted. The report CSV file is

correctly created, with the reported message and reason being recorded in the file.

```
1 User message,Bot response,Report reason
2 "Hey, what did you have for lunch today?",I love french toast! It's one of my favorite foods. What kind did you have?,This is a test report
```

Finally, the theme switcher works on all pages, with all pages correctly switching theme, keeping the same information on the page, including the data in the Jinja variables, and the same page loaded.

Stakeholder testing

Now that the program has all the intended features added and the program has been properly tested to the point where I am relatively sure there are no bugs, I can allow stakeholders to test the program. The stakeholder will be given a link to the GitHub page, from where they can download and install the program using the instructions provided. They will then be asked to fill in a questionnaire in order to provide feedback.

Questions to ask.

1. How did you find the installation process?
 - Whilst not directly related to the performance of the program or how it meets the stakeholder's expectations, whether a user can actually download and install the program with relative ease is important, as if it is too difficult to download, users will not want to use it at all, regardless of the program's actual performance.
2. How easy was it to understand how to use the chatbot?
 - In a similar vein to the previous question, whether users can actually understand how to use the chatbot is an important aspect. This question should be for the tester to talk about how easy it is to enter messages into the chatbot, and follow the instructions presented on the home screen. It should also allow the tester to talk about any issues or bugs that appear when talking to the chatbot.
3. How appropriate were the chatbot's messages? Did you feel like it understood your sentence, and provided a coherent response?
 - This is to test the main part of the program – the chatbot itself. This question should help me to understand how the chatbot performs in real world situations with a wide variety of different messages that may be unlike any that I have tested it with. This question is perhaps the most important as whether the chatbot can generate messages that the user finds appropriate and understandable is the make-or-break part of the program. For the program to work as intended, the user should find the responses natural, otherwise they may not feel as comfortable or as interested in talking to the chatbot.
4. What topics did you talk to the chatbot about, and where were any topics you felt the chatbot was weaker in?
 - This is another way of testing the efficacy of the chatbot, and how it performs under a wide variety of different topics and conversations. This question may help to pinpoint weaker areas of the algorithm, so that I can improve it later down the line.
5. Did you use the report feature?
 - This question can test how useful the report feature has turned out to be. It can also test how easy it was to understand and use, as if the tester has not used it, it may be because they didn't know it was there or didn't understand how to use it. If they did use it, it is important to get their feedback on how useful they felt it was, whether they felt as though it gave them enough information, and enough reporting options.

6. Did you use the search feature?

- This question can test how useful the search feature turned out to be. It can also test how easy it was to understand and use. Did the tester adequately understand how to search for messages, and understand what the search results were? If they did not use it, why not? Did they not know it existed, or did they find that it wasn't that useful. Also, did they find that the search suggestions were helpful?

Question responses.

1. How did you find the installation process?

- Fairly easy and simple, instructions were good as the language was simple which made it easy to understand.

2. How easy was it to understand how to use the chatbot?

- Very easy and simple, that chat bot was very engaging in the conversation.

3. How appropriate were the chatbot's messages? Did you feel like it understood your sentence, and provided a coherent response?

- Yes, it felt like speaking to a real human being. It gave very human-like responses. For example, I asked what kind of salad it liked, and it described an actual salad that was known to me.

4. What topics did you talk to the chatbot about, and where there any topics you felt the chatbot was weaker in?

- When I expressed that his opinions were interesting, he didn't know what to say, and so came up with something sort of random. Other than that, when we spoke about salad and lunch, it was good, and he could engage in a good, long conversation.

5. Did you use the report feature?

- Nope, as there was never any need to, all of the bot's responses were appropriate, so there was no point in reporting anything.

6. Did you use the search feature?

- I used it, it highlighted the text which made it easy to spot, it was very simple to use and convenient. The autocorrect suggestions were good as it meant I could see what I wrote and didn't have to remember myself. It is very efficient.

Analysis

The installation process is clearly easy enough for users to understand. This is very important as it removes the first barrier to entry. Whilst obviously the more important aspect is that the program works well, user's may be turned away from the program before even trying it if they cannot install it and access it easily enough, so therefore the fact that they can is very good.

The feedback on the quality of the chatbot's responses is good and shows that the chatbot is performing as expected. The fact that the chatbot struggled with being told that its opinions are interesting is expected, as I suspect that the message simply said something along the lines of 'that is interesting', which is a message the bot is expected to struggle with, as due to the fact that the bot only takes as input each individual sequence, and therefore will not be able to understand input sequences that refer to previous messages. This is a known weakness of the bot, and something that could be worked on at a later stage.

Whilst it would be good to get more in-depth feedback on the reporting feature, it is a good thing that the bot performs good enough for none of its messages to be reported.

The feedback on the search feature is good, as whether users could properly understand how the search feature worked, and how to understand the results page, was a possible point of concern, so it is good that it is easily understandable.

The stakeholder clearly enjoys the program, and finds that it fits its purpose properly, which is good feedback, and shows that the program is clearly fit for purpose.

Evaluation

Success criteria review

Refer to the Stakeholder requirements – success criteria section on pages 16 and 17 for the detailed success criteria.

- **Intuitive, simple interface.** As you can see in Figure 81, the app has a simple interface, similar to those found in most text messaging platforms like WhatsApp. The program uses individual message bubbles for each message with a different colour for messages sent by the user and messages sent by the bot. There is a box at the bottom to enter messages, along with a submit button. The other features such as the search function, the report function and the theme switcher are also obvious and easily accessible. Therefore, this success criteria has been met.

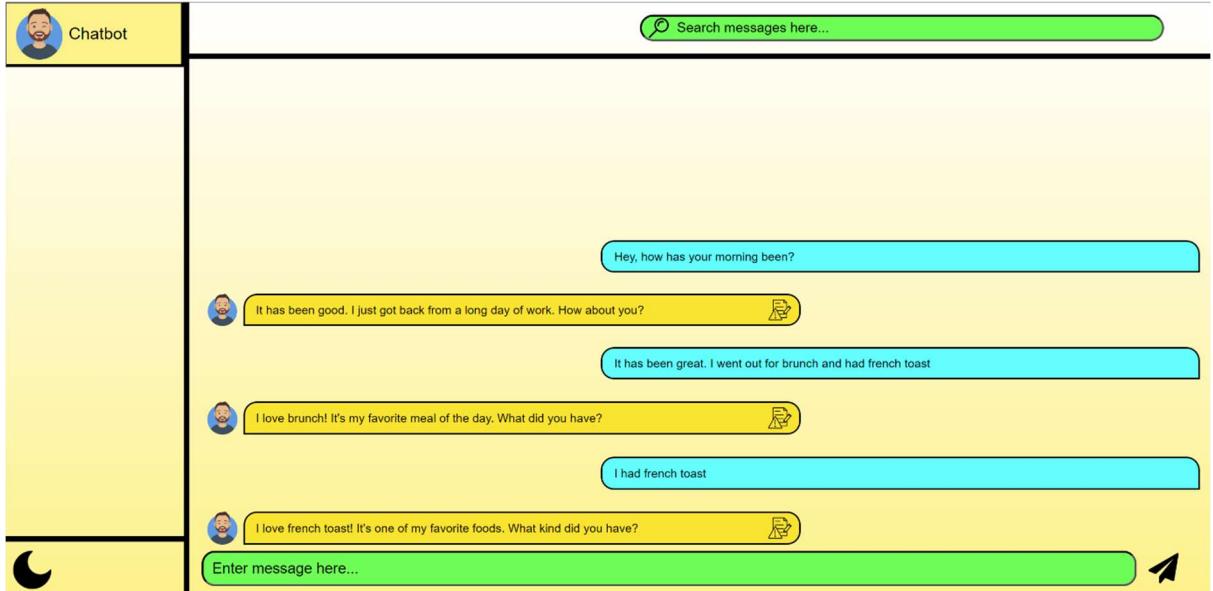


Figure 81

- **Customisable display.** Below shows the two available themes, both a light mode and dark mode. These give the users different options as to the appearance of the webapp, and the changed theme is used throughout the app. Therefore, this success criteria has been met.



Figure 82

- **A working AI model.** As shown in Figure 83, the program contains a machine learning model capable of holding a good, understandable conversation with the user. Whilst it's not completely perfect, making the occasional slip-up in understanding the user's message, it is

more than good enough, and stakeholder testing showed that users should find it good enough. Therefore, this success criteria has been met.

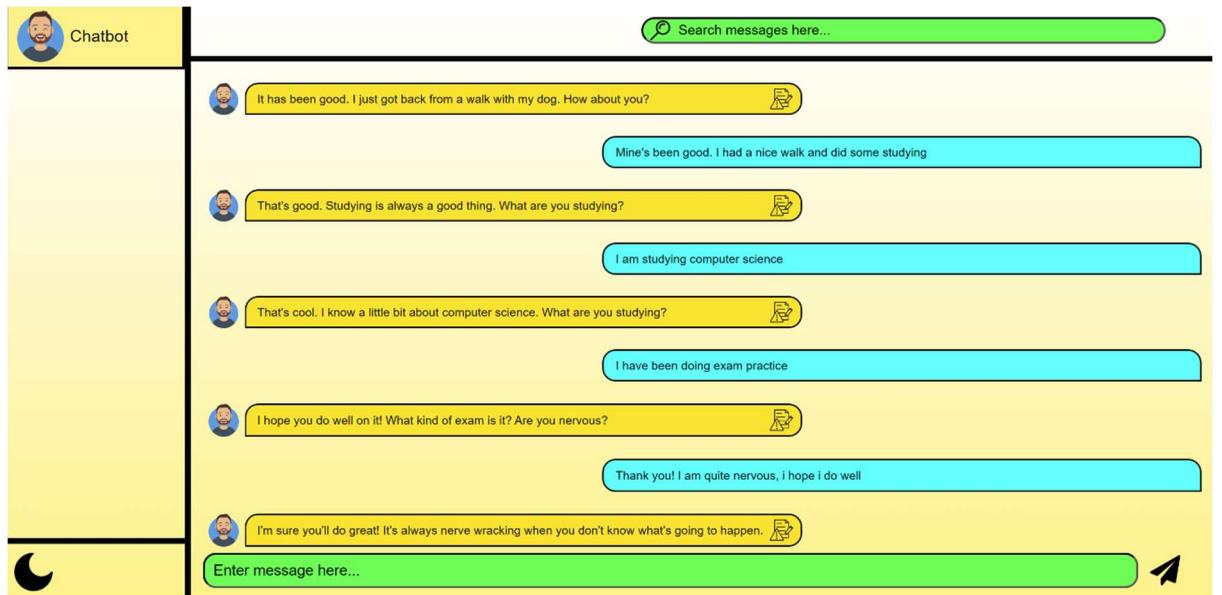


Figure 83

- **Search feature.** The program has a search feature, with a search bar in the top right of each page (shown in blue in Figure 84**Error! Reference source not found.**). The search bar includes a drop-down menu (shown in red in Figure 84) that automatically adjusts whilst the user is typing to suggest messages the user could be searching for. When the user submits their search, a new page renders showing the searched-for message, highlighted in a different colour, alongside a selection of messages before and after to give context and show the part of the conversation it is from. Therefore this success criteria has been met.

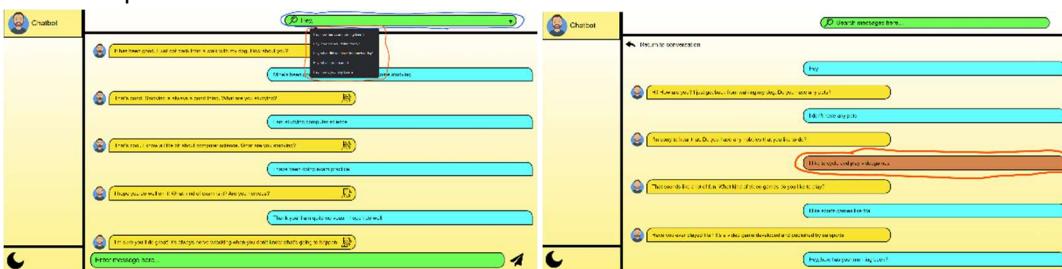


Figure 84

- **Reporting.** Circled in green in Figure 85, there is a report button on the end of every AI message bubble, which opens a pop-up box giving the user the ability to enter a reason for reporting the message. Submitting the form not only renders a page thanking the user for reporting a message, but also saves the reported message to a separate CSV file, along with the previous user message and the report reason. However, the generation time of the message is not also logged in the CSV file, as this was impossible to get with the method I used for getting the reported message. I don't believe this is a massive issue, as if the computation time was too long, users would most likely report the message with that as the report reason. Therefore, this success criteria has mostly been met.

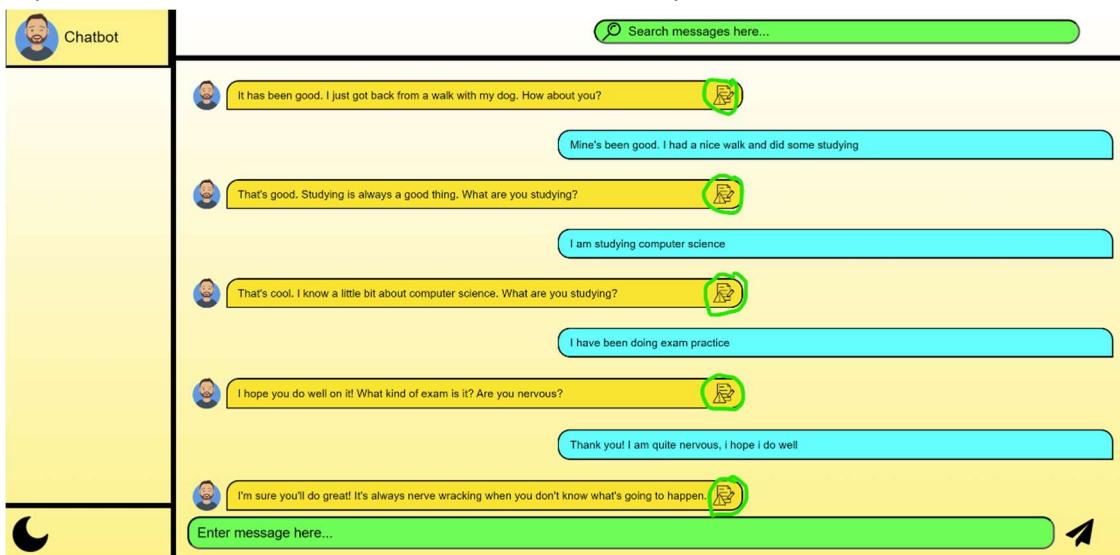


Figure 85

- **Speech-to-text.** This feature proved too difficult to add in the development time available. Not only did it run into a number of complex errors, but the process of training and fine-tuning the model for automatic speech recognition may have been impractical anyways, due to the computer hardware I have available. Whilst the fine-tuning process isn't strictly necessary, without it the speech recognition model would have likely been very inaccurate, and so wouldn't have been worth adding without it. Therefore, this success criteria has not been met.

All features in this program have been thoroughly tested to ensure they work in all use cases, with iterative testing and quality control done throughout development, as well as the entire program being tested from start to finish after development, by both me and a stakeholder, who verified that it met their expectations. The program was also destructively tested, ensuring that all input fields are capable of handling unexpected inputs, such as nonsense text entered into the message input box.

Limitations

One major limitation of the program is, as mentioned earlier, the lack of speech-to-text input. Not having this feature means that there is only one method for the user to input messages into the program, which can also make the program worse for accessibility or any kind of hands-free use.

Another limitation is to do with the actual response generation. The response generation model only takes as input the sentence the user just entered into the message submit box, and not the rest of the conversation. This means that the bot is always lacking context, making it impossible for it to understand phrases or words that refer to previous messages, making each message have to be explicit in its meaning instead of referring to other parts of the conversation. This can make the conversation feel more structured, and less natural.

A further limitation with the program is the processing power required. This is a fundamental drawback with this type of program, particularly due to the heavy reliance on machine learning, and is impossible to really overcome.

How to avoid these limitations

With more development time, speech-to-text input could be added. The algorithms for recording, saving and reading in audio were already created and tested in development, what is missing is actually training and implementing the automatic speech recognition model. With more development time and available computation power this may be possible.

Implementing this would give the user more options for sending messages to the bot, and an option that may be more natural and allow a freer conversation, mostly hands-free.

Adding context to response generation is also possible. I could read in a few previous messages from the log file, which would be the last few messages in the conversation. These messages could then be added to the newly inputted message and inputted into the model as a combined input. This would mean the model would have the context of the conversation and would be able to understand more implicit parts of the input sentence, that refer to other parts of the conversation.

Overcoming the limitation of high processing power requirements is not really possible, as the algorithms and models used will need a lot of processing power to generate responses. The only possible way to ensure that this wouldn't affect the user would be to host the program on a separate server, and have users access it online. However, this is an impossible solution for me, due to the costs involved in running or renting a server, so for now this limitation will remain.

Future development

As mentioned before, an important feature to add in the future would be contextual message generation, by loading previous messages from the log file, and then inputting them into the model alongside the new messages. This would allow there to be a more free-flowing conversation, with users able to be more implicit with their messages and not just specifying the main point of the conversation repeatedly.

As the program is modular, with features added using a combination of separate functions, adding any new features that the stakeholders may want should be easily done by simply adding on new functions, or swapping out older ones with updated versions. As the program is split across many files, with separate files for each web page, and for each styling, adding new stylings or pages, or tweaking current ones should be easy.

Also, as the code is well commented and annotated, with docstrings and type hints for each function, any new developer should easily be able to understand the code behind the program and be able to add to it themselves.

Future versions of the program could incorporate new features such as chatbots with different personalities. This could be done by fine-tuning the model on a dataset of a certain person's conversation (such as from a movie or video game script), and then saving that model separately. I could then add it into the sidebar of the webapp, underneath the current chatbot icon, allowing users to easily switch between them, perhaps saving the conversation history when the user switches chatbot.

Another possible addition could be splitting the extended chat log by conversation, so that when showing search results, only messages from the correct conversation would be shown.

Final code

GitHub - https://github.com/Kaziksobo/NEA_Chatbot

app.py

```
 1 import flask
 2 from flask import Flask, render_template, request
 3 from os import remove, path
 4 from app_functions import (
 5     reply_generator,
 6     log, log_reader,
 7     message_id_generator,
 8     format_message,
 9     message_selector,
10     create_log_file,
11     log_report,
12     get_messages_list,
13     english_check
14 )
15
16 current_theme = 'light'
17 current_page = 'index.html'
18 stylesheet = 'static/light-styles.css'
19 chat_history = []
20 messages_to_display = []
21 query_message = ''
22 reported_message = ''
23
24 app = Flask(__name__)
25
26 @app.route('/')
27 def main() -> flask.Response:
28     """Displays the home page 'index.html' and clears temp logs"""
29
30     global stylesheet, current_page
31
32     # Deletes and recreates main log file
33     if path.exists('log.csv'):
34         remove('log.csv')
35     create_log_file('log.csv')
36     if not path.exists('ext_log.csv'):
37         create_log_file('ext_log.csv')
38
39     # Render home page with appropriate stylesheet and datalist
40     current_page = 'index.html'
41     return render_template('index.html', stylesheet=stylesheet, messages_list=get_messages_list())
42
43 @app.route('/home', methods=['GET', 'POST'])
44 def home() -> flask.Response:
45     """
46
47     Displays the home page 'index.html' and clears temp logs \n
48     This is the same as the main() function, however is used for when the home button is pressed
49
50     """
51
52     global stylesheet, current_page
53
54     # Deletes and recreates main log file
55     remove('log.csv')
56     create_log_file('log.csv')
57
58     # Render home page with appropriate stylesheet and datalist
59     current_page = 'index.html'
60     return render_template('index.html', stylesheet=stylesheet, messages_list=get_messages_list())
61
62 @app.route('/message', methods=['GET', 'POST'])
63 def message() -> flask.Response:
64     """Requests the user's message, processes it and renders the messages page with the user's message and AI's reply"""
65
66     global stylesheet, current_page, chat_history
67
68     # Requesting the data entered into the input form (the search bar) by the user
69     user_message = request.form['message-input']
70     user_message = format_message(user_message)
71
72     if english_check(user_message):
73         reply, time = reply_generator(user_message)
74     else:
75         reply = 'That makes no sense'
76         time = 0
77
78     log(
79         user_message=user_message,
80         bot_response=reply,
```

```

79     time_taken=time
80   )
81
82   # Reading in chat history from log file
83   chat_history = log_reader('log.csv', format=True, Len_Limit=True)
84
85   # Generating Ids for chat messages to order them with CSS ids
86   chat_history = message_id_generator(chat_history)
87
88   # Render message page with chat history, appropriate stylesheet and datalist
89   current_page = 'message.html'
90   return render_template('message.html', stylesheet=stylesheet, messages=chat_history, messages_list=get_messages_list())
91
92 @app.route('/theme', methods=['GET', 'POST'])
93 def theme_switcher() -> flask.Response:
94   """Switches the theme of the web app"""
95
96   global current_page, current_theme, stylesheet, chat_history, query_message, reported_message
97
98   # Switch stylesheet and theme
99   if current_theme == 'light':
100     current_theme = 'dark'
101     stylesheet = 'static/dark-styles.css'
102   else:
103     current_theme = 'light'
104     stylesheet = 'static/light-styles.css'
105
106   # Render current page with corrected stylesheet
107   if current_page == 'message.html':
108     return render_template(current_page, stylesheet=stylesheet, messages=chat_history, messages_list=get_messages_list())
109   elif current_page == 'search_result.html':
110     return render_template(current_page, stylesheet=stylesheet, messages=messages_to_display, messages_list=get_messages_list())
111   elif current_page == 'search_error.html':
112     return render_template(current_page, stylesheet=stylesheet, message=query_message, messages_list=get_messages_list())
113   elif current_page == 'report.html':
114     return render_template(current_page, stylesheet=stylesheet, message=reported_message, messages_list=get_messages_list())
115   else:
116     return render_template(current_page, stylesheet=stylesheet, messages_list=get_messages_list())
117
118 @app.route('/search', methods=['GET', 'POST'])
119 def search() -> flask.Response:
120   """Renders searched message plus number of surrounding messages"""
121
122   global messages_to_display, current_page, stylesheet, query_message
123
124   # Use text input to get query message
125   query_message = request.form['search-bar']
126
127   print(f'Search query - {query_message}')
128
129   # Get full chat history and reverse
130   history = log_reader('ext_log.csv', format=True)
131   history.reverse()
132
133   # Find Location of query message in message history
134   location = False
135   for message_info in history:
136     if message_info['text'] == query_message:
137       print(f'Found message - {message_info}')
138       location = history.index(message_info)
139
140   # If the message could not be found in the history, an error page is rendered, which references the query message
141   if not location:
142     current_page = 'search_error.html'
143     return render_template(current_page, stylesheet=stylesheet, message=query_message, messages_list=get_messages_list())
144
145   # Selects the number of messages to show before and after the query message
146   messages_before, messages_after = message_selector(len(history), location)
147
148   # Uses messages_before and messages_after to create a List of messages to show to the user
149   # selects messages from the history with indexes between (location + messages_before) and (location - messages_after + 1)
150   messages_to_display = [history[i] for i in range((location + (messages_after)), (location - (messages_before + 1)), -1)]
151
152   # Generating Ids for chat messages to order them with CSS ids
153   messages_to_display = message_id_generator(messages_to_display)
154
155   # Labels the query message to allow it to have different CSS
156   for message in messages_to_display:
157     if message['text'] == query_message:
158       message['class'] = 'query-message-box'
159     else:
160       message['class'] = False
161
162   # Render search results page with the List of searched messages to show, appropriate stylesheet and datalist
163   current_page = 'search_result.html'
164   return render_template(current_page, stylesheet=stylesheet, messages=messages_to_display, messages_list=get_messages_list())
165

```

```

166 @app.route('/back', methods=['GET', 'POST'])
167 def back() -> flask.Response:
168     """Renders the message page when back button on search results/search error/report page clicked"""
169
170     global chat_history, current_page, stylesheet
171
172     # Render message page with chat history, appropriate stylesheet and datalist
173     current_page = 'message.html'
174     return render_template(current_page, stylesheet=stylesheet, messages=chat_history, messages_list=get_messages_list())
175
176 @app.route('/report', methods=['GET', 'POST'])
177 def report() -> flask.Response:
178     """Logs the report and renders report page"""
179
180     global stylesheet, chat_history, current_page, reported_message
181
182     # Finds the name of the form used to report the message
183     # The name of the form contains the id of the message
184     reported_message = list(request.form.to_dict().keys())[0]
185     # Get the report reason from form
186     report_reason = request.form[reported_message]
187     # Get message id from report form name
188     report_message = reported_message.replace('report-', '')
189
190     # Finds the message with the matching message id as to the one from the report form name
191     for message in chat_history:
192         if message['id'] == report_message:
193             location = chat_history.index(message)
194             bot_response = message['text']
195             reported_message = bot_response
196             # Finds the message before the reported message
197             user_message = chat_history[location - 1]['text']
198
199             log_report(user_message, bot_response, report_reason)
200
201             # Render report page with reported message, appropriate stylesheet and datalist
202             current_page = 'report.html'
203             return render_template(current_page, stylesheet=stylesheet, message=bot_response, messages_list=get_messages_list())
204
205     if __name__ == '__main__':
206         app.run(debug=True, port=5000, host='0.0.0.0')

```

app_functions.py

```
1  from transformers import BlenderbotTokenizer, BlenderbotForConditionalGeneration
2  from csv import writer, reader
3  from os import path, remove
4  from time import time
5  from datetime import datetime
6  from typing import Union
7  from nltk.corpus import words
8  from nltk import word_tokenize
9  import torch, re, nltk.data, psutil
10
11 def log_reader(file_address: str, format: bool=False, len_limit: bool=False) -> list:
12     """Reads the contents of the Log file into a single List, ignoring the time taken and Log time columns.\n13     Also has the option of formatting the chat history, turning each message into a dictionary,\n14     specifying if the message came from the model or the user"""
15
16     # If chat history does not exist, function returns none
17     if not path.exists(file_address):
18         print('No log file found')
19         return None
20
21     # Function continues here if the chat history exists
22     print('Log file found')
23     with open(file_address, 'r') as log_file:
24         csv_reader = reader(log_file)
25         chat_history = []
26         # Create a List of messages in chat history
27         for row in csv_reader:
28             # Ignores header row
29             if 'User message' in row:
30                 continue
31             # Appends the first two entries in the row to the chat_history List
32             chat_history.append((row[0], row[1]))
33
34     # Reverses the List so that the most recent messages are first (new - old)
35     chat_history.reverse()
36
37     if format:
38         chat_history = format_chat_history(chat_history, len_limit)
39
40     return chat_history
41
42 def format_chat_history(chat_history: list, len_limit: bool) -> list:
43     """Formats messages in chat history in form:\n44     {
45         'type': 'ai' or 'user',
46         'text': message content
47     }"""
48
49     formatted_chat_history = []
50     # If the length of the chat history should be limited, the length will be limited to 9 messages
51     # This will be done by only putting up to the first 9 messages into the new formatted List
52     # 9 is chosen as it is the number of available message bubbles on the display
53     if len_limit:
54         for i in range(min(9, len(chat_history))):
55             # Messages with even indexes are AI messages, whilst ones with odd indexes are user messages
56             if i % 2 == 0:
57                 formatted_chat_history.append({'type': 'ai', 'text': chat_history[i]})
58             else:
59                 formatted_chat_history.append({'type': 'user', 'text': chat_history[i]})
60     # If the length of the chat history should not be limited, the entire history is formatted
61     else:
62         for i in range(len(chat_history)):
63             if i % 2 == 0:
64                 formatted_chat_history.append({'type': 'ai', 'text': chat_history[i]})
65             else:
66                 formatted_chat_history.append({'type': 'user', 'text': chat_history[i]})
67
68     return formatted_chat_history
69
70 def message_id_generator(chat_history: list) -> list:
71     """Adds IDs to each message based on their location in the List\n72     Each message in the chat history should now be in the form:
73     {
74         'type': 'ai' or 'user',
75         'text': message content,
76         'id': between 'message-1' and 'message-9'
77     }"""
78
79     for i in range(len(chat_history)):
80         chat_history[i]['id'] = f'message-{i + 1}'
81
82     return chat_history
83
84
85 def reply_generator(message: str) -> Union[str, float]:
86     # sourcery skip: extract-duplicate-method
87     """Uses Blenderbot to generate a reply to the inputted message"""
88
89
90     # The exact time at the start of the message generation process is saved
91     start = time()
92     name = 'facebook/blenderbot-400M-distill'
```

```

93     print(f'Input sequence: {message}')
94
95     # Declares tokenizer
96     # Checks specified cache directory for tokenizer, if tokenizer not present it will be downloaded
97     tokenizer = BlenderbotTokenizer.from_pretrained(name, cache_dir='data/tokenizers')
98
99     print('Tokenizing input')
100
101    # Tokenizes input, adding special tokens and returning PyTorch tensors
102    input_ids = tokenizer.encode(
103        message,
104        add_special_tokens=True,
105        is_split_into_words=False,
106        return_tensors='pt',
107    )
108
109    print(f'Input Ids: {input_ids}')
110
111    reply_ids = model_generation(name, input_ids)
112
113    # Decodes tokens from model, avoiding special tokens so they do not appear in the input
114    reply = tokenizer.decode(reply_ids[0], skip_special_tokens=True)
115
116    print(f'Response: {reply}')
117
118    # The exact time at the end of the message generation process is saved
119    end = time()
120
121    # Returns message, and the time taken is calculated by subtracting the exact time at the start from the time at the end
122    return format_message(reply), round((end - start), 2)
123
124
125 def model_generation(name: str, input_ids: torch.Tensor) -> torch.Tensor:
126     """Uses BlenderBot to generate a response sequence to the inputted tokens, with beam search for text generation"""
127
128     # Declares model
129     model = BlenderbotForConditionalGeneration.from_pretrained(name, cache_dir='data/models')
130
131     num_beams = beams_calc()
132
133     print('Generating reply ids')
134     # Generates Reply ids using beam search to generate text
135     result = model.generate(
136         input_ids,
137         num_beams=num_beams,
138         max_length=60
139     )
140     print(f'num_beams used: {num_beams}')
141     print(f'Reply Ids: {result}')
142
143     return result
144
145 def beams_calc() -> int:
146     """Calculate num_beams to use in model generation based on CPU frequency.
147     Currently flips between 35 beams in above 3.4GHz, otherwise uses 1 beam"""
148
149     cpu_freq = psutil.cpu_freq()
150     cpu_freq = cpu_freq.max
151
152     return 35 if cpu_freq > 3400 else 1
153
154 def format_message(message: str) -> str:
155     """Formats message to capitalise and remove whitespace and fix some grammar errors"""
156
157     # Fixes formatting of full stops, with no space before and one space after
158     message = re.sub(r'\s(?=[.,;:])', "", message)
159     # Removes any whitespace from the start and end of the message
160     message = message.strip()
161     # Capitalises any bare 'is'
162     message.replace(' i ', ' I ')
163     # Uses NLTK to split the message into a list of sentences
164     sent_tokenizer = nltk.data.load('tokenizers/punkt/english.pickle')
165     sentences = sent_tokenizer.tokenize(message)
166     # Capitalises each sentence and joins it back together into one string
167     sentences = [sent.capitalize() for sent in sentences]
168     message = ' '.join(sentences)
169
170     return message
171
172 def english_check(message: str) -> bool:
173     """Checks if the message is understandable english by tokenizing it into words, and checking if at least one of them is in the dictionary"""
174
175     # Used NLTK word tokenizer to split the message into a list of words
176     words_list = word_tokenize(message)
177     return any(word.lower() in words.words() for word in words_list)
178
179
180 def log(user_message: str, bot_response: str, time_taken: float) -> None:
181     # sourcery skip: extract-method
182     """Logs the user's message, the bot response and the computation time to a CSV file"""
183
184     # Checks if both of the log files exists, creating them if they don't
185     file_address = 'log.csv'
186     extended_file_address = 'ext_log.csv'
187     for file in [file_address, extended_file_address]:

```

```

188     if not path.exists(file):
189         create_log_file(file)
190
191     # Checking if the main log file is full
192     with open(file_address, 'r', encoding='utf-8', newline='') as file_object:
193         # As the limit is 10 message-response pairs, this checks if the file is more than 11 rows long (10 pairs and the header row)
194         limit_reached = sum(1 for _ in file_object) == 11
195     if limit_reached:
196         print('Log file limit reached')
197
198     # Reading in all rows from original file
199     rows = []
200     with open(file_address, 'r', encoding='utf-8', newline='') as log_file:
201         csv_reader = reader(log_file)
202         rows.extend(iter(csv_reader))
203
204     # Removing oldest message row and header row (the first two rows read into the list)
205     rows.pop(0)
206     rows.pop(0)
207     # OS.remove is used to delete the file
208     remove(file_address)
209     create_log_file(file_address)
210
211     # Writing older rows to new log file
212     with open(file_address, 'a', encoding='utf-8', newline='') as file_object:
213         csv_writer = writer(file_object)
214         for row in rows:
215             csv_writer.writerow(row)
216
217     # Logging new data to both log files
218     print('Logging data')
219     # Storing the exact time the data is being Logged
220     log_time = str(datetime.now())
221     # Logging the data into both log files
222     log_file_writer(user_message, bot_response, time_taken, file_address, log_time)
223     log_file_writer(user_message, bot_response, time_taken, extended_file_address, log_time)
224
225 def log_file_writer(user_message: str, bot_response: str, time_taken: str, file_address: str, log_time:str) -> None:
226     """Logs data to any log file"""
227
228     # Writes a row of data into the specified file
229     with open(file_address, 'a', encoding='utf-8', newline='') as file_object:
230         csv_writer = writer(file_object)
231         csv_writer.writerow([user_message, bot_response, time_taken, log_time])
232
233 def create_log_file(file_address: str) -> None:
234     """Creates the log file, adding in the header row"""
235
236     # Attempts to open the file in write mode which will create the file if it cannot be found
237     print('Creating new log file')
238     with open(file_address, 'w', encoding='utf-8', newline='') as file_object:
239         csv_writer = writer(file_object)
240         # Write the header row to the newly created, empty file
241         csv_writer.writerow(['User message', 'Bot response', 'Time taken', 'Log time'])
242
243 def log_report(user_message: str, bot_response: str, report_reason: str):
244     """Saves reported response, user's message and report reason to a CSV file"""
245
246     # If the file does not exist, it is created
247     if not path.exists('report.csv'):
248         create_report_file()
249
250     # Writes a row of data to the report CSV file
251     with open('report.csv', 'a', encoding='utf-8', newline='') as file_object:
252         csv_writer = writer(file_object)
253         csv_writer.writerow([user_message, bot_response, report_reason])
254
255 def create_report_file() -> None:
256     """Creates report file, adding in the header row"""
257
258     # Attempts to open the file in write mode which will create the file if it cannot be found
259     print('Creating report file')
260     with open('report.csv', 'w', encoding='utf-8', newline='') as file_object:
261         csv_writer = writer(file_object)
262         # Write the header row to the newly created, empty file
263         csv_writer.writerow(['User message', 'Bot response', 'Report reason'])
264
265
266 def message_selector(length: int, location: int) -> int:
267     """Selects the messages to be displayed when a message is searched for,
268     based off the length of the chat history\n
269     Up to 9 messages should be displayed, with the number before and after the query varying
270     depending on the position of the query message in the list and the length of the list.\n
271     This function is necessary to ensure that the algorithm never wraps round
272     and starts displaying messages from other parts of the list"""
273
274     messages_before = 0
275     messages_after = 0
276
277     # Calculates the number of messages available in the list on either side of the query
278     messages_after_query = length - (location + 1)
279     messages_before_query = length - (messages_after_query + 1)
280
281     # If there is at least 4 messages available on either side of the query, then 4 messages will be displayed from either side of the query
282     if messages_after_query > 3 and messages_before_query > 3:
283         messages_after = 4
284         messages_before = 4
285
286     # If there are less than 4 messages available after the query, then all messages available to be displayed after the query will be,

```

```
285     # and an appropriate number of messages will be displayed before to ensure that there are 9 messages shown in total
286     elif messages_after_query < 4 and messages_before_query > 3:
287         messages_after = messages_after_query
288         messages_before = 8 - messages_after
289     # If there are less than 4 messages available before the query, than all messages available to be displayed before the query will be,
290     # and an appropriate number of messages will be displayed after to ensure that there are 9 messages shown in total
291     elif messages_after_query > 3 and messages_before_query < 4:
292         messages_before = messages_before_query
293         messages_after = 8 - messages_before
294
295     return messages_before, messages_after
296
297 def get_messages_list() -> list:
298     """Returns a List of the text content of all the messages, in a List with each element containing a message"""
299
300     history = log_reader('ext_log.csv', format=True)
301     history.reverse()
302
303     return [message['text'] for message in history]
304
```