

Python Chatbot

Contents

Analysis:

- ____ Problem Identification - **Page 3**
- Stakeholders - **Page 4**
- Why Is It Suited To A Computational Solution? - **Page 5**
- Computational Methods That The Solution Lends Itself To -

Pages 6-7

- Interview - **Pages 8-12**
- Existing Similar Solutions - **Pages 13-16**
- Features Of The Proposed Solution - **Page 17**
- Further Meeting With Stakeholders - **Pages 18-19**
- Requirements - **Pages 20-23**
- Success Criteria - **Pages 24-25**

Design:

- ____ User Interface Design - **Pages 26-27**
- Usability Features - **Pages 28-29**
- Stakeholder Input - **Page 30**
- Algorithms - **Pages 31-35**
- Subroutines - **Pages 36-46**
- Diagrams To Show How The Subroutines Link - **Pages 47-49**
- Inputs And Outputs - **Page 50**
- Key Variables - **Pages 51-52**
- Validation - **Page 53**
- Testing Method - **Pages 54-57**

Development & Testing:

Stage 1: Training Data Creation - **Pages 58-71**

Stage 1 Review - **Pages 72-73**

Stage 2: Bot Training - **Pages 74-100**

Stage 2 Review - **Pages 101-102**

Stage 3: Response Creation - **Pages 103-108**

Stage 3 Review - **Pages 109-110**

Stage 4: Main User Interface - **Pages 111-134**

Stage 4 Review - **Pages 135-136**

Stage 5: Final Testing - **Pages 137-141**

Stage 6: Stakeholder Testing - **Pages 142-144**

Evaluation:

Criteria Met - **Pages 145-155**

Usability Features - **Pages 156-157**

Survey - **Pages 158-164**

Limitations - **Pages 165-166**

Maintenance - **Pages 167-168**

Bibliography - **Pages 169-170**

Final Code:

TrainingData.py - **Pages 171-173**

NumericalData.py - **Pages 174-175**

Training.py - **Page 176**

Model.py - **Pages 177-178**

UserInterface.py - **Pages 179-182**

Analysis

Problem Identification

Software-specific chatbots have been available in computer software for many years, ranging from Google Assistant to Apple's Siri. This list even includes custom website chatbots for client support on business websites. However, there has been a limited amount of chatbots with its primary purpose being for user entertainment via chatting as well as chatbots for educational purposes. The chatbot which I will be creating in this project will be able to be used for English language studies. Stakeholders for the educational side may include students whose English is not their first language and students with minor learning disabilities, which hinders their written communication skills. The educational aim is to help these people have a more natural time communicating using the English language. This chatbot could also be used for leisure.

The requirement to complete this solution is for the bot to be able to use natural language manipulation to understand and respond to messages sent from the user via a user interface. One of the main problems is getting the chatbot to understand a word from the user when, in context, it may have a different meaning. Another problem is that there are several varieties of the same word. An example of this is in different tenses, plural, singular, etc. This makes it difficult for the bot to understand what words are used due to having to know hundreds of variations of words. I can, however, overcome this problem by using processes such as stemming or lemmatization, which will break down words, in whatever tense and context, into their roots.

Stakeholders

The clients and demographics for this software would be casual software users, which are going to represent the leisure part of this chatbot, as well as students with minor learning disabilities (representing the educational section). A casual software user is anyone who is knowledgeable in computer software and is capable of testing the bot for its leisure purposes. Meanwhile, the students with minor learning disabilities will be testing the bot for how well it helps the user to learn from their spelling mistakes by the bot correcting them.

Many people, often while travelling on public transport, find it hard to do anything entertaining or even anything to do to pass the time. A chatbot with its unique personality traits available to be used could thus bring leisure to these types of people. My stakeholder for this category is Ella Tysar, who is familiar with both computer software and creative mindset. She will be testing the chatbot's entertainment aspect.

You can also use this chatbot for support in the English language. For example, when a person learns by seeing how to spell day-to-day words that are often misspelt and being corrected when they make mistakes. My stakeholder for this area is Gabriella Bishop. She is a dyslexic student who often struggles in her daily life how to spell commonly used words. By interacting with this chatbot, she will be able to be supported with her spelling by watching the bot spell words and phrase sentences correctly as well as correcting her spelling mistakes.

Why Is It Suited To A Computational Solution?

This problem lends itself to computational methods of finding and implementing a solution for a variety of reasons. The solution will be software with a graphical user interface that is capable of running a chat-based conversation between the user and the chatbot. The chatbot should be able to interpret and accurately respond to the user's messages on this user interface, according to what their message contains. This program is required to run on a computer as the graphical user interface is modified and created intended to be seen using a monitor. It also involves user input via a keyboard and for the user to use a mouse to click buttons represented on the user interface. There are alternatives to this solution that do not involve a computer. However, they include forms of one, for example, a phone or tablet. The aspects of the solution allow for the user to enter a message and send it to the chatbot where the bot will then respond to the user graphically. This will be done using a form of a text message in a chat-based environment between the bot and the user. A computer would be able to solve this problem better over a human because a person can make mistakes, especially when it comes to written conversation. Often people find themselves misspelling words whereas a computer would always have the correct word spellings. For education in the English language, a computer would do a better job as it's unlikely to make as many grammatical or spelling mistakes than another person, especially since a large number of people struggle with English grammar and use a lot of slang in their conversations (which can be filtered out if using a chatbot).

Computational Methods That The Solution Lends Itself To

Problem Recognition

The main problem is finding a way to allow the chatbot to understand a set of strings (the words in the messages) so that it will be capable of responding to them. An underlying problem is finding a way for the bot to be able to recognise a set of words, take out the unnecessary words in the message and use the relevant words to compose a response. To compose the response, I will need to use a form of training data to train my chatbot model suitable responses to different types of messages. Once the bot knows how to form a response, the rest is setting a graphical user interface in which the user can send the message and read replies while also having access to the entire conversation before exit. With the potential of the bot being able to correct user mistakes, the bot must spell check before providing a general response to the message so that errors are corrected.

Problem Decomposition

This central problem can be decomposed into a set of much smaller, more straightforward steps. The following is an initial idea for what these steps are:

1. The chatbot receives text input from the user.
2. Stop words and punctuation in the user's message are taken out.
3. The edited message is checked for spelling mistakes.
4. Response specific to the user's message or a response to correct a user mistake is created.
5. The answer from the chatbot is displayed to the user.

These five steps present the cycle of how the program will work. It will continue as a loop as long as there is a conversation between the bot and the user. If the user has sent no message or exited the program, this cycle ends at step 5.

Divide And Conquer

Breaking down this project into smaller steps makes it easier to combine solved solutions into an entire program. Even though these individual steps are challenging, they are still doable. Breaking this problem down into smaller steps refers to the divide and conquer method of problem-solving. I can break this problem into five parts: the user interface, the analysis of the user's message, the training data, training the chatbot to speak from the training data and the

response creation. The analysis of the user's message relies upon the user interface as it requires the message sent in the input box. The response creation relies upon the analysis of the user's message as it will use the most decomposed form of the user's message to create a reply. The response creation is also dependent on the training of the chatbot so the program can create its own responses. The training data and the training of the chatbot are linked as this training data is the data which the chatbot will use to learn suitable responses to different types of messages.

Abstraction

When the chatbot receives a message from the user, the message will contain a lot of unnecessary data. An example of this is punctuation. The chatbot does not need to learn or understand how punctuation works so keeping it in the message will overcomplicate the code. Another example is the use of non-root words for example “cook”, “cooking” and “cooked” all have the same main meaning in the context of food. The chatbot will need to return a response according to context so if all words are turned into their original root word, it would be much easier for the chatbot to determine the most suitable response to the user.

Interview

Interview Questions

I will outline some of the critical questions to be asked of each stakeholder. The questions will find out what features they are looking for in a chatbot application.

Educational

My questions for Gabriella, who represents the educational side of the bot, are as follows:

1. Have you found that reading is a helpful way to learn English?
2. Have you ever had an experience with using a chatbot?
3. If yes, what is your background with chatbots and what features of that particular bot did you like?
4. How do you think a chatbot will help you with developing your English skills such as spelling?
5. How would you like the software to be operated?
6. What features would you like to see in a new chatbot?
7. Do you have anything else to add?

Questions one and four identify what she thinks are useful features that the bot could use to help a person learn English better. These can be used when programming how the bot will function.

Questions two and three ask for her previous experience with chatbots which can help with determining features that make a chatbot useful and helpful so that it can be used for supporting English education.

Questions five and six are more related to how the bot will be presented and the way that it should be designed. Like questions two and three, question six is related to features that will help make the bot present itself better.

Interview - Gabriella

1. Have you found that reading is a helpful way to learn English?

"I think that reading enhances my English skills as I become accustomed to reading words over again. This develops skills such as spelling and grammar, which is especially important to me as I am attempting to battle my Dyslexia."

2. Have you ever had an experience with using a chatbot?

"Yes, I have previously used Online chatbots for entertainment purposes. However, I have not experienced a chatbot that helps me with English development skills, so I think that this would be an interesting project."

3. If yes, what is your background with chatbots and what features of that particular bot did you like?

"Although I do not remember the name of the chatbot which I last used as it was many years ago, I remember that it was entertaining because it had funny replies to the text that I typed. I liked that feature of the avatar along with the bot's reply. This made for an entertaining experience that kept me intrigued."

4. How do you think a chat bot will help you with developing your English skills such as spelling?

"I think that it could help as long as the chatbot is correct in its use of words, this way it would get into my head and I could subconsciously remember them."

5. How would you like the software to be operated?

"Most of the chatbots that I used many years ago were web-based, so I think it would be cool if it were different so that I get a different experience. For this reason, I think it's best to have the chatbot as an application."

6. What features would you like to see in a new chatbot?

"I would like the chatbot to be comprehensive. As well as this, I think it would be a great idea if the chatbot can reply specifically to what I have said so that it feels like it understands what I have said and it behaves as if we were having a conversation."

7. Do you have anything else to add?

"I think the bot would be more enticing if the program that it's based on were simplistic as I don't think it needs to be too complicated to complete its function. I also think that it would help me learn better if the bot can correct any spelling mistakes that I make in my responses so that I can learn to spell better. To remember and go over my common spelling mistakes, it would be

useful for me to have a copy of the session so that I can go over and practice my spelling when not using the program.”

Analysis

My stakeholder finds it helpful to read words to learn them, which is useful as the bot is designed to communicate with the user via text. Furthermore, she had previous experience with chatbots thus was able to recommend having the bot able to respond to subject-specific queries and suggesting that the bot should correct any spelling mistakes so that it is able to educate her better by allowing her to remember the correct spellings. An interesting suggestion from Gabriella is for the ability to save the conversation. This can be done in a text file when clicked on a button.

Leisure

My questions for Ella, who represents the leisure side of the bot, are as follows:

1. Have you ever found communication as a form of entertainment?
2. Have you ever had an experience with using a chatbot?
3. If yes, what is your background with chatbots and what features of that particular bot did you like?
4. How do you think a chatbot will help you pass the time or entertain you?
5. How would you like the software to be operated?
6. What features would you like to see in a new chatbot?
7. Do you have anything else to add?

Questions one and four identify what she thinks of communication (such as in the case of a chatbot) and if it can be considered as a form of entertainment which can affect the way that the chatbot works.

Questions two and three ask for her previous experience with chatbots which can help with determining features that make a chatbot useful and helpful so that it can be used for entertainment.

Questions five and six are more related to how the bot will be presented and the way that it should be designed. Like questions two and three, question six is related to features that will help make the bot more presentable.

Interview - Ella

1. Have you ever found communication a form of entertainment?

"Yes, I read books regularly as entertainment. However, they tend to be dramatised and descriptive books. I also chat on the phone via text messages which I guess is a form of communication which is entertaining."

2. Have you ever had an experience with using a chatbot?

"No, I have never used one before."

3. If yes, what is your background with chatbots and what features of that particular bot did you like?

N/A

4. How do you think a chatbot will help you pass the time or entertain you?

"I will be able to talk to the bot just like a normal person in a text conversation which can pass the time. This will be entertaining to me if the bot can respond to my messages accurately instead of assuming what I said or responding several times similarly. In this way, it will seem like I am talking to a person and not necessarily a robot."

5. How would you like the software to be operated?

"I would like for there to be a big screen where you can easily get started with a conversation without having to know too much about the program. I don't want to waste too much time learning how to use the program, and I would rather get straight into it."

6. What features would you like to see in a new chatbot?

"I don't want there to be too many buttons like when entering my messages. I should be able to press enter when I am done typing instead of pressing a submit button each time because that will make sending messages easier and faster for me. I believe that a chatbot should be very straightforward to use."

7. Do you have anything else to add?

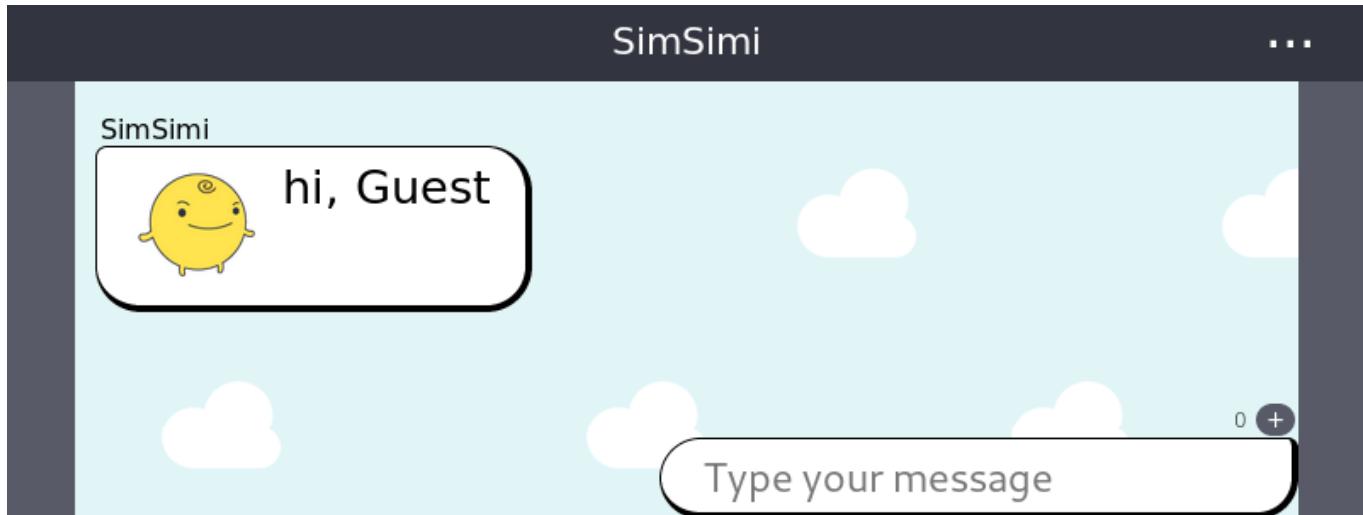
"For the user interface to be as simple as possible would be perfect for me. Any useful features such as refresh the conversation to start a new one without having to close and re-open the program."

Analysis

This stakeholder has never used a chatbot before, but she does find that reading text is a form of entertainment for her. Furthermore, she stated that she would enjoy talking to the chatbot to pass the time and as entertainment. She has requested for the bot to be able to respond in a machine learning manner (so the replies will be specific to her messages) and for the user interface to be simplistic/easy to use. She also suggested for me to add a feature which refreshes the conversation and vaguely mentioned any other “useful features” should be added.

Existing Similar Solutions

SimSimi

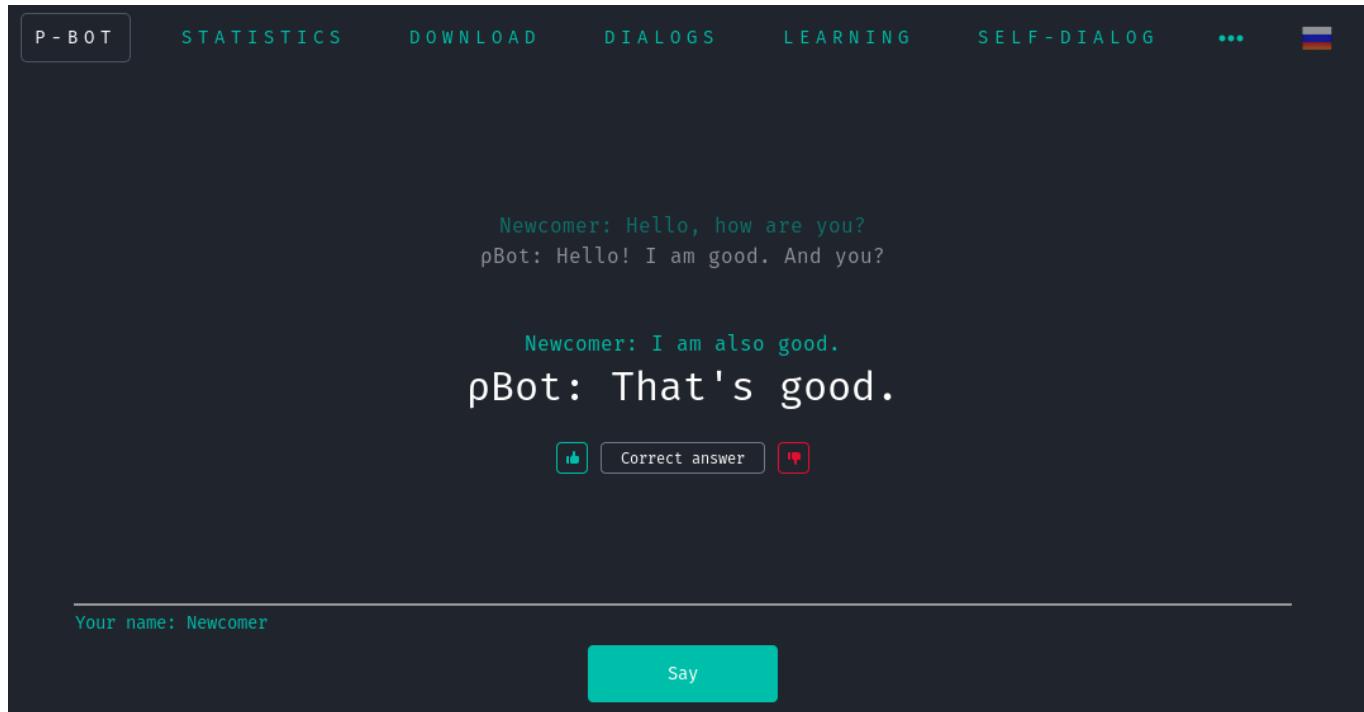


SimSimi is a free-to-use chatbot which is both web-based and found as applications on operating systems such as iOS. This program allows you to talk to a bot by sending messages through its user interface. This user interface is relatively complex, but it mainly focuses on the messages sent between the bot and the user. The user interface contains settings which change aspects of the program as well as settings for the chatbot. It also allows you to add responses for specific messages if said by you or another user using this chatbot. For example, "How are you?" could be responded by "Meow" if that's what the user set as the response. As much as this idea would sound cool to have in a bot, for this particular bot, I found it inconvenient. This is because, while talking to the bot, it often returns with inappropriate messages due to the community contribution affecting the chatbot's responses. However, the bot does have a scale setting for how much the bot can swear. With this feature, some of the inappropriate comments from the bot won't be seen.

Parts that I can apply to my solution:

Even with the community features, the bot has its personality (although it may frequently change due to the community features). Though I don't plan on adding these community features, this is mainly because my bot will be program-based and not web-based. Also, the personality of the bot may vary according to how it constructs the messages from the database. I like how it is laid out similar to a conversation or a text chat such as how they are displayed on mobile phones. This feature has inspired me to create a similar design. However, I plan to have a user interface that will be more simplistic (e.g. my program might not have a fancy background visible to the user) as I want the user to focus on the conversation rather than everything around it.

pBot



pBot is an online chatbot which runs on its website. You can also download this bot if you wish to, so it can work as an application. pBot works similarly to the example above as this bot can reply to messages correctly and can also learn how to respond correctly. However, here, it's according to the user's ratings of its responses. The ratings are available using the green thumbs up button, the red thumbs down button and the "Correct answer" button on the user interface. This means that this chatbot is capable of learning communication skills from the user. However, this chatbot is different to my plan as I do not want to implement a learning feature that is accessible by the user. The user will not be able to adapt the bot's behaviour towards them. Instead, I want to train my chatbot using machine learning so that it can construct its suitable responses. pBot works with several different languages but my bot will only be available in English.

Parts that I can apply to my solution:

In a pBot conversation, you can only see a fixed amount of messages (four) before they disappear from the user interface. I think this would cause a problem if this were a feature in my chatbot. This is because, for the educational aspect, if the chatbot corrects the user early in the conversation and the user wants to look back at it, they would be unable to. This means that it limits the chatbot's ability to teach the user their mistakes, as when forgotten, it can't be seen again unless they make the same mistake again. Instead, I will apply a conversation area that

stays during the entire chat session between the bot and the user. This chat area will contain a scroll bar which the user can use to scroll to earlier messages.

Botsify



Botsify is a chatbot which helps automate messages for people/companies. It can create conversation forms, acquire user form comments and be used for: websites, Facebook, Slack and education. It aims to provide faster customer support, increased customer retention and satisfaction; break the language barrier and more. This bot can be set up using a configuration panel given to you where you can configure the bot's behaviour. It can also reply to queries from the users/customers and respond to them, providing them with the help that they need.

Parts that I can apply to my solution:

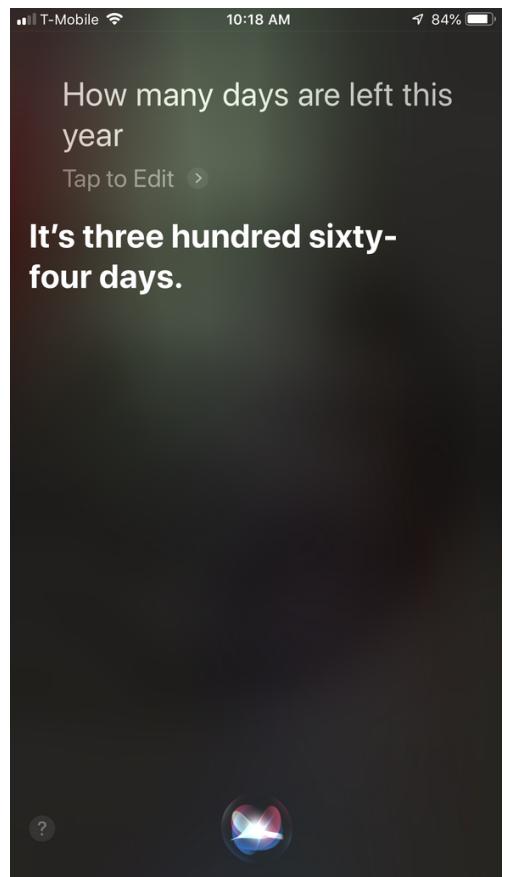
This chatbot is very different from the one that I am planning as its primary purpose is to provide support for clients of businesses. However, it can reply to customer queries precisely, which is the part that I would like to use in my solution. For example, the bot in my solution will not only be able to respond to statements but also answer subject-specific questions such as “What’s your favourite movie” or “What’s your favourite food”. This would be different from Botsify as they may have pre-coded answers to frequently asked questions rather than use machine learning to respond to questions.

Siri

Siri is a virtual assistant that is part of Apple's iOS operating system. This program uses voice queries and a natural-language user interface to answer questions, make recommendations and perform actions. The software adapts to the users' language usages, searches and preferences. Siri supports a wide range of user commands such as performing phone actions, checking necessary information, scheduling events and reminders; searching the Internet and others.

Parts that I can apply to my solution:

Siri is a virtual assistant that does not apply to my solution as, although it uses queries and answers questions (with this being done using voice), it aims to answer to the user's needs and assist them with tasks. My solution does not intend on using voice queries, and it will function as a chatbot to talk to the user rather than support them like Siri. However, Siri has given me the idea of instead of using voice queries to listen to questions; it should be able to read out the chatbots messages to the user. This allows for anyone with seeing disabilities to still interact with the bot by listening to it instead. Similar to Siri, it will be able to suggest the correct word when misspelt words are found in user messages.



Features Of The Proposed Solution

Initial Concept Of My Solution Considering This Research

My solution will be an application that when started, will bring up its user interface. When the application has begun, the chatbot will reset its chat to the default (empty) and will wait for the user's first message. The user will be able to use the user interface to communicate with the program. This will be done by inputting their responses to the bot's messages as well as the ability to read the bot's responses in a conversation-like way. As the user speaks to the bot, the bot will reply with relevant responses to the user's queries using machine learning. The user will be able to quit at any time by clicking the exit button.

Limitations Of My Solution

The main limitation of my solution is that the bot would not be able to understand the user. An example of this is when the user makes several spelling mistakes in their messages. This would be because the program would not recognise the word; therefore, the bot may not be able to respond correctly. This could further result in the user being unable to be supported in English by the chatbot as well as it should. However, one of my stakeholders has suggested that I make the bot able to correct spelling mistakes. This means that the bot will not be able to assume what the user intended to say. With this suggested feature, the bot will correct the user instead of responding to the statement that was said. This prevents the chatbot from assuming what the person said and meant and then creates a response to the assumption.

Another limitation of my solution is the bot's responses. Because the chatbot is using machine learning to learn how to respond to the user's message from training data created from a movie script corpus, it may respond weirdly to the user's messages e.g. very dramatically as it learnt responses from conversations in movies. Also, I can make my chatbot better by teaching it more responses but creating more responses as training data and teaching the chatbot these responses will take up more time. This creates the limitation even using the Cornell Movie-DIALOGS Corpus, which has over 220,000 conversations, will take a lot of time. A possible solution to this is using Reddit messages which offer far greater topics, more realistic forms of communication between people and contains millions of conversations. The problem with using Reddit though is that the chatbot may be more prone to speak in slang or vulgar language, which would compromise its teaching ability. Filtering this out may be difficult as inappropriate language may be written in different forms and a filter would have to be applied against millions of messages.

Further Meeting With Stakeholders

The Following Is An Email That I Sent To My Stakeholders:

"Hey,

I have been looking at different ways to present the chatbot as an application, and I have come up with some ideas which I would like your input on. The first way is to split the application into three sections: 2 sections where there's a bot response followed by a user response and a part where the user can type their input. When a new reply is sent, whether it is from the user or the bot, the section will fill with the response. The sections will always be filled if the different sections were filled earlier. For example, if a new message from the user and a reply from the bot were sent and the top was filled, the next time this happens, the bottom section will be filled and vice versa until the session ends. Therefore this method works by replacing text in existing sections. The other way is to portray the chat like a conversation (such as in applications like Whatsapp). However, this could be split into two versions: when the page gets full, the top messages are deleted or keep all messages in the conversation and allow the user to use a scroll bar to scroll through previous messages that were sent in this current session.

Thanks,

Danielle."

Gabriella:

"Dear Danielle,

I like the idea of using a conversational-based design. I think this will make for a more delightful experience and it will be easy and straightforward to use. I think it would also be better if you could keep all of the previous messages, so the user can scroll through them, just as a text, which is a part of the conversation design anyway. This is because I feel like I may forget some of the spelling mistakes that the chatbot has already corrected so I may want to look back at it and remember.

Gabriella."

Ella:

"Dear Danielle,

Danielle Kushnir

I didn't really understand what you were explaining in the first section of the email, but I do like the idea of the application working like a casual chat (or Whatsapp as you suggested in the 2nd part). However, I do want to be able to see the difference between my messages and the chatbot so that I know the difference while chatting to not get confused.

Best regards,

Ella.”

Requirements

Hardware

A computer capable of running the software with a monitor – The software will be displayed on an application, thus will need a machine that can view it.

Keyboard – Some form of a keyboard for text communication is required due to the application containing a user input box. This will be used to collect the user's responses.

Computer mouse – Some form of a mouse is required so that the user is capable of using the software's buttons and clicking on the input box to type a message. To use the scrollbar, the user will need to use a mouse.

CPU – Starting with Tensorflow 1.6 (I am using Tensorflow 2), binaries use AVX instructions which may not run on older CPUs

Software

Python Interpreter - All of the code is written in Python; thus, a Python interpreter is required.

Python Virtual Environment – Tensorflow recommends to install Tensorflow and other modules that it will use in a virtual environment. Even though it's only recommended, I found that Tensorflow only works on my computer with a virtual environment so it may be required on some computers.

Operating System – There are several operating system requirements for Tensorflow: Ubuntu 16.04 or later (64-bit), macOS 10.12.6 (Sierra) or later (64-bit), Windows 7 or later (64-bit) and Raspbian 9.0 or later

NLTK for Python - This is the natural language library that I will be using. It will allow the bot to manipulate the language used in the user's messages so that it can interpret it and reply.

ImportLib for Python - This library opens other Python modules, which makes it easier to use code from external modules written by myself.

String for Python - For easy removal of punctuation from strings.

Tkinter for Python - This provides the user interface.

JSON for Python – The training data used will be stored in the form of JSON.

PyEnchant for Python – This module is used to check for the spellings of words as well as suggesting corrections for those spellings.

Tensorflow for Python – Tensorflow is a module which I can use to create a neural network

TFlearn for Python – TFlearn sits on top of Tensorflow and will also be used to create the chatbot model

Numpy for Python – The neural networks uses numbers so I will need to use numpy for the conversion of the training data to numbers

Pickle for Python - Used to save data structures that can then be loaded in other modules.

pyttsx3 for Python - Used for text to speech when the chatbot responds to the user.

Stakeholder Requirements

Design

Requirements	Explanation
Simple design	The design needs to be simple and not confusing for the user to use.
Conversation feed with scroll bar	So that the user can see past messages during the current conversation and scroll through them.
An obvious user input section	This is so that the user can communicate with the chatbot simply and so that it's clear where the user can start.
An avatar for the bot	This gives further personality to the bot.

Functionality

Requirements	Explanation
A user input section	This is so that the user can communicate with the chat bot by sending them messages which are to be typed in this text box.
A submit function	Used to enter user input from the text box so that the chat bot is able to receive, 'read' and reply.
Some form of message box	All messages sent by either the user or the bot will be in some form of message box. These are displayed so that the user can see the message that they typed as well as the bot's responses. The main requirement here is that the user is able to see all messages as well as having the ability to differentiate between their own messages and the chatbot's messages.
Avatar	The avatar will be displayed at the top of the screen so that the user can see who they are talking to.
Main menu/screen	A main menu that contains a section where the chat between the user and the chatbot will occur and the user input section.
Saving a conversation	Some kind of function that allows a user to save their current conversation into a file so it can be reviewed by the user at a later time.

Hardware And Software

Requirements	Explanation
Standard peripherals: computer with a keyboard, mouse and monitor	The user needs a basic computer to use this chat bot. The computer must have standard items such as storage and a processor. The file size will be less than a gigabyte.

Python with all of the required libraries	The program will run on Python with libraries such as Tensorflow, NLTK and more. Tkinter will be used for the user interface. There are modules with smaller uses in the code like JSON which are needed.
Virtual environment	Tensorflow works best in a virtual environment, it sometimes may not work at all if there's no virtual environment.
Windows, Mac or Linux operating system	These are the operating systems supported by Python.

Success Criteria

Requirements	How To Evidence
Simple design	Screenshot of the window with large sections and the messages are not overlapping. Feedback from the user with timings showing how long it took them to understand the interface.
Option to press a button to stop the program	Screenshot of the exit button in the user interface as well as code to show how it works.
Clear message box which shows the chat between the user and the bot	Screenshot showing the chat section with sample input messages.
Text area for input	Screenshot of the user input section as well as code to show how the chat bot is using this input to show that it works as well as how pressing enter works instead of a submit button.
Bot avatar	Screen shot of the user interface showing the chat bot's avatar.
The bot is able to respond with a relevant reply	Screenshot of an example of a bot's response to a user's message and the code to show that it works.
Training data creation	Screenshot of the code creating the training data as well as what the data is stored as.
Training of the chatbot	Screenshot of the code which uses the training data to train the chatbot as well as a screenshot showing the output and process of the training.
Use of machine learning for responses	Screenshot of the code showing how the chat bot responds to the user's messages.

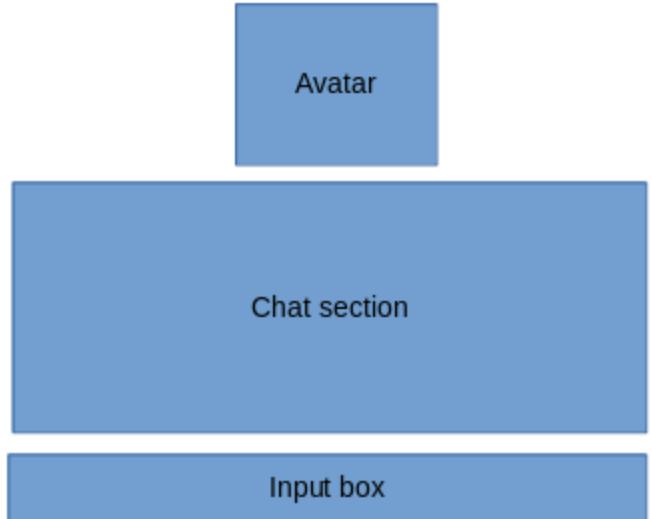
The first five requirements are success criteria for the design of the program and the user interface. The stakeholders have requested for the program to be easy to use hence it has to have a simple design. In order to find out whether these requirements have been met, I could

run a test on different people to experiment how easy to use the user interface is. They could grade the simplicity of the user interface and according to their grade, it would affect whether this section has been successfully completed. A success is if the user interface is found to be simple and easy to use. The last four requirements are related to the chatbot's ability to respond to the user contextually. The requirements "The bot is able to respond with a relevant reply" and "Use of machine learning for responses" will be successful if the chatbot is able to compose a relevant response to the user. I could test this by selecting 10 phrases and if the majority of the responses are contextually correct, these requirements would be marked as successful. The requirement "Training data creation" would be successful if the training data has successfully been converted to the format required. The requirement "Training of the chatbot" would be successful if the neural network is created using the training data created with no provided errors.

Design

User Interface Design

To the right is the design for the user interface. On this user interface, there is a chat section and an input box which allows the user to interact with the chatbot. The chat section is continuous as it contains all of the messages sent by the chatbot and user, displayed like a conversation. The user will be able to scroll through messages throughout their communication with the bot to see their message history for their current session. There is also an avatar section for aesthetics reasons; an image breaks some of the simplicity of the user interface. A feature that's not seen in the diagram are the application menus such as "File" and "Settings" which will contain small features and some configurations for the user.



Links to the success criteria:

- Simple design

User Input Box

The user input box is the area where the user can chat with the chatbot. Here they will type their messages. This user input box requires a way for the user to enter their input which would be done by either pressing a key or a submit button. After submission, the input will then be forwarded and analysed by the bot. Before this occurs, the message will be displayed in the chat section along with a user prefix so that the user can identify who sent the message. After the user's message is displayed, the user input box will be cleared.

Links to the success criteria:

- Text area for input

Chat Section

The chat section is the area of the user interface where the user can see their conversation between themselves and the chatbot throughout their session. Each message in this section will include a prefix, identifying the message as being sent by either the user or the chatbot. Any message sent by the user through the user input box will be displayed here as well as responses created by the chatbot. There will be a scrollbar which the user can use to scroll through their existing session to see their message history. Previous sessions are not saved because after the program closes, the chat section resets so the user can not access previous conversations. This means that I would have to allow the user to store the conversation somehow so it can be saved to fulfill my stakeholder's request.

Link to success criteria:

- Clear message box which shows the chat between the user and the bot.

Avatar

The avatar section appears at the top of the user interface, on top of the chat section. This section contains an avatar image for the chatbot. This section has been added for aesthetics as the stakeholders may prefer to see an image of "who" they are talking to so they feel like the conversation is less automated. Also, my stakeholder Gabriella has stated that she liked the avatar feature on chatbots from her previous experience of using chatbots during her interview. Thus, she has suggested for it to be added as a feature for this chatbot.

Link to success criteria:

- Bot avatar

Usability Features

The following are the usability features included in order to make sure that the program is easy to use for any user. For the design, all sections are large to make it easy to see, especially the chat section since that is the main area that the user will be using.

Any text on the window needs to be large enough and easily readable for people with vision difficulties. This is especially important for the chat and user input sections as those are the areas where the user needs to be able to read a large amount of text. If the user is unable to read their messages while they are entering new inputs, they may make mistakes in their message. This may result in the chatbot being unable to understand what the user has said and thus correcting them. However, they will be corrected for a mistake that was not their fault or given a corrected spelling for a word which they know how to spell. In case of the scenario of people using screen readers to read text out loud for them, I have to ensure that the text on the program is considered as actual text and not images or Tkinter objects to ensure that the screen reader works. I may decide to implement a feature that reads out messages for the user so that they do not have to use a screen reader. The best solution for this seems to be to provide my own text to speech feature. Furthermore, the font used in the chat section has to be basic because fonts can make it harder to read as some people/screen readers may not be able to differentiate the letters used.

The design of the main window shows large sections and clear text; however, the final version will have a different colour scheme. In order to keep the user interface simple, I have decided to adopt a white colour scheme on the user interface with a limited amount of colours to prevent the user interface from being overcomplicated. If this colour scheme changes, it is crucial that the text will still be visible.

The usage of the application needs to be easy. This involves things like not having too many buttons or options which change features in the program. There is only one menu (the main menu which opens when the user first runs the application) which reduces the risk of the user getting lost in the program. Another reduction for the possibility of the user getting lost is the fact that the menu is divided into three sections with these sections being large. Moreover, keeping the usage of the application to be easy prevents the creation of guidelines or tutorials where the user will have to spend time learning how to use the program, which also leaves room for confusion.

The main focus here is the communication between the chatbot and the user. This communication has to be capable of running without being over complicated (for the user). I can overcome this by using the most suitable message as the chatbot's response. Contextual responses allow for the user to understand who they are talking to better as well as feeling like

they are talking to someone rather than an automation (or predefined responses).

Link to success criteria:

- Clear message box which shows the chat between the user and the bot

Stakeholder Input

I Sent My Stakeholders The Following Email:

"Hi,

Please find attached a diagram of how the chatbot application will look like. I have made sure to include the conversational-based design as to your suggestions. I am planning on creating a mood system for the bot, and I would like your input on this. Moods for the chatbot means that its replies to your messages will change according to what mood it is in, which will change according to what you have told it. Moreover, the chatbot's avatar will change to represent its current mood, for example, smiling to show that it is currently happy.

Thanks,

Danielle."

This Is The Reply I Have Received From Gabriella:

"Hey Danielle,

The diagram that you sent me representing the user interface looks very simple. I am sure that as you develop the application, the user interface will look a lot better, for example, adding buttons or the actual layout of the design. The mood idea sounds cool but might not work well when asking a question. For example, asking what its favourite colour is and the bot replies with how happy it is instead. Also, I don't like the idea that the bot might ignore me by saying something related to its mood rather than replying to my message.

Gabriella."

This Is The Reply I Have Received From Ella:

"Dear Danielle,

I like the design that you sent for the user interface. The application looks very easy to use, as requested. However, I do not like your mood idea because I feel like it would respond with moods in situations where it should not, so it would not work well. For example, if I do not say things that trigger a bot mood change, nothing will change or if I make a spelling mistake but instead of the chatbot correcting me, it will say how angry it is. Also, I would not want to use a chatbot which could harass me during our conversation when angry.

Best regards, Ella."

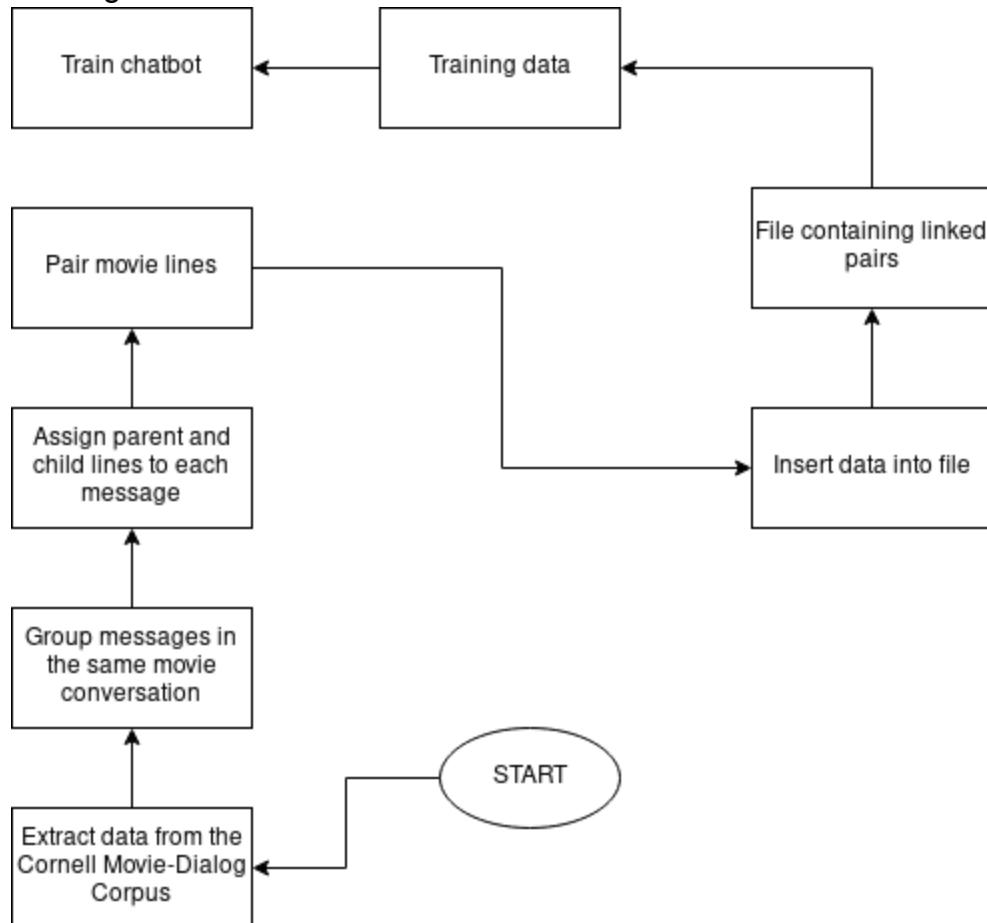
Algorithms

Other than the user interface and the message analysis parts of this solution, the main challenge lies in creating teaching data then teaching the chatbot to speak using that data. The overall objective for this is:

Extract data from a message database → Pair extracted data (parent and child messages) → Teach chatbot to speak using the pairs

This can be broken down into two linked sections, creation of the training data and the creation of the model to be used (this will involve teaching the model using the training data). These can then be broken down further into smaller problems. Two flowcharts can be used to illustrate how this process will work.

Training Data Creation:



The process of creating training data can be split into two: manipulating the data extracted from the Cornell Movie-Dialog Corpus into suitable data and data storage.

Data Manipulation:

- Extract lines from corpus
- Sort lines into their correct movies
- Sort lines into their correct conversation
- Pair up parent and child lines
- Format the lines into suitable data (for example, remove punctuation)

Data Storage:

- Create a storage file (text or JSON)
- Insert data into the file to be stored

Using this data, I can train the chatbot during the creation of the model.

Each step in more detail:

Data Manipulation:

Extract lines from corpus:

Cornell's Movie-Dialog Corpus stores a lot of information in several text files regarding movie information, conversations, movie lines and more. All of this information is great however I only need information regarding the movie lines for this project. Therefore, I will need to extract the lines from the movie lines text file so that I only use this part of the corpus. The corpus has made it easy to extract data by providing tags which separate variables.

Sort lines into their correct movies:

There are over 200,000 conversations stored in the corpus so I can assume that there are hundreds of thousands of lines in movies. However, these lines don't all come from the same movie. Because I want to pair up a parent and child message, they have to be from the same conversation and movie thus I need to ensure that they come from the same movie first (as they won't be in the same conversation if they aren't in the same movie). The corpus helps this process as it stores the unique movie ID in the movie lines text file next to each line so the movie they come from can be identified.

Sort lines into their correct conversation:

Like I said above, for a line to be a response (child) to the line before (the parent), they have to be said after each other as lines are stored one after the other. However, what if the next line is in a different part of the movie? As in, a different conversation. That means the child is not actually the child of our current parent. This means that I must check if they occur in the same conversation before pairing up messages. The movie lines text file stores all the lines in each row so I can imply that the line underneath is most likely the child of the current line and I can infer from looking at some of the data stored in the file that the movie lines are stored sequentially to what happens in that particular scenario. This means I can always take the line below and get its conversation ID. If the conversation IDs of the parent and the child lines match, they are indeed parents and children of one another.

Pair up parent and children lines:

This part is the easiest because after the two prior checks are done, I can simply add these to an array or set variables. These variables would be a parent variable as the current parent line and a child variable as the current child line. These would change for every set of parent-children links. In order to store the same pairs together, I will need to link them for example storing them in an array and appending it to a larger 2D array which contains other paired messages.

Format the lines into suitable data:

Punctuation is a big (but easy to deal with) problem that I have to deal with when storing the training data and comparing a user's input. This is because, when the chatbot model is making comparisons, I don't want it to classify "Hello there!" and "hello there." as two separate messages due to the capitalisation and punctuation used. When saying both of these phrases, the user should be expecting similar responses (not necessarily the same response but it should be similar context-wise) from the chatbot, without the punctuation affecting its similarity calculations. This process of formatting would include removing common keyboard punctuations such as bullet points, commas and changing "don't" to "dont" by removing the punctuation. I could create a list or dictionary full of punctuation that need to be filtered out and removed from a line/message.

Data Storage:

Create a storage file:

My two options for the storage files are a text file or a JSON file. There is no major advantage that I could find against the other but I did find that the JSON file will be able to store the data in a better format. This is because all of the data will be stored in dictionaries where I could have arrays of information per paired message for example the parent and child of this paired message. Also, I don't have to store each piece of data on a separate line or separate them out using tags (similarly to how the corpus has done it) like I would need to in a text file. This could save disk space, especially because I am going to be dealing with relatively large file sizes due to the great amount of paired up messages. Because storing data in a JSON file seems to fit my requirements better than a text file.

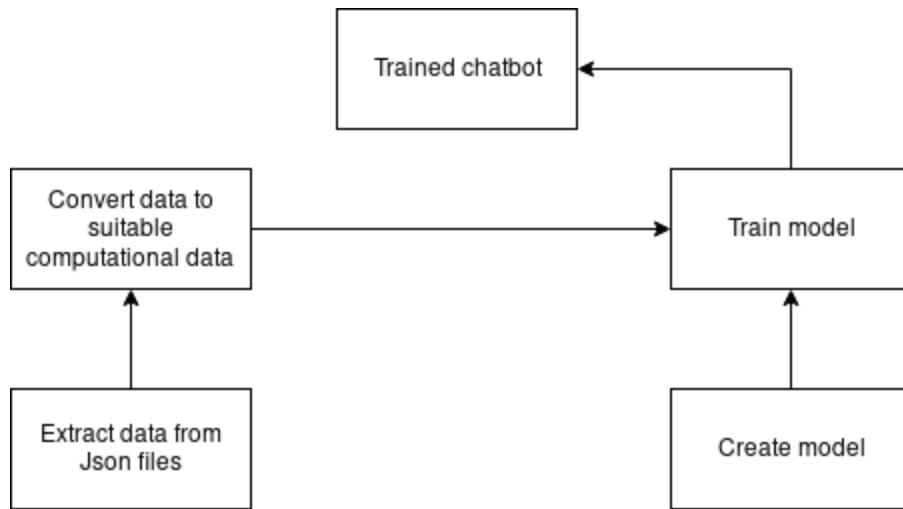
Insert data into file to be stored:

Using a JSON file, it's very easy to insert (or 'dump') data into the file for storage. The main requirement for this is a 1D array which consists of all the data being inserted (mainly the parent and child). With this array, I can 'dump' the data into the JSON file and refresh the array, so it can be used for the next paired messages. Another possible option is using a large 2D array which will store all of these 1D arrays which could be 'dumped' all at the same time. The JSON file will store a dictionary filled with information gained from an array and placed into the format of a dictionary before insertion.

Link to success criteria:

This entire process of creating training data is a requirement in my success criteria. Completing this process fulfills the requirement.

Training:



This training process can be split into two smaller processes: data conversion to numbers and creation of the chatbot model.

Data Conversion:

- Extract paired messages from the file, keeping them linked
- Create a numerical bag of words using extracted data

Creation of The Chatbot Model:

- Create a model
- Train the model using the bag of words

Data Conversion:

Extract paired messages from the file:

My training data has been formatted as a very large dictionary in a JSON file. To extract the information that I need from this file, I will need to use a for loop, which will loop through every single paired message and turn them into a bag of words before moving on to the next one. I will need to keep a list of all of the unique words in the file (with no duplicate words) so that my model can identify words written from a user input to help with comparing the most probable parent said and construct a response for the user.

Create a numerical bag of words:

Neural networks work with numbers thus I will need to adjust my training data so it's suitable for use. I will be using the module NumPy for creating numerical arrays and other number manipulations which are required for the neural network. In my bag of words, I will represent a "0" as the word not being present and a "1" as present in that position. In this way, I will later be able to use a bag of words to identify (using the same method) whether a word is present and where so that the model can use this for analysis. The bag of words itself will be an array filled with a combination of "0"s and "1"s.

Creation of The Chatbot Model:

Create a model:

For creating my chatbot model, I will be using the module Tensorflow. This will allow me to create a machine learning chatbot. Using Tensorflow, I can create a neural network. The key part of creating the model is the settings that I configure before I train the chatbot. I can later play around with these settings to find the right options that make the bot work better for my training data.

Train the model:

Using the neural network I just created using Tensorflow and the bag of words (my training data) that I created earlier, I can train my model to recognise certain patterns and link them together to form the best possible response for the user. This training will be done using the module TFlearn. TFlearn sits just above Tensorflow and provides a way of training neural networks. After this process is done, I will have a working model which is capable of detecting patterns to create context thus allowing the identification of the best possible response in the training data that could be used to respond to the user's query.

Link to success criteria:

- The bot is able to respond with a relevant reply
- Training of the chatbot
- Use of machine learning for responses

Subroutines

Now that I know how my model will work, I can plan the subroutines that this project will contain. The program will be split into four files, the user interface, training data, chatbot model and user response creation. Keeping these sections in separate files makes my project more organised. The main files that will link together are the chatbot model with the user response creation and the user interface with the user response creation. I can easily call functions from the other files when it needs to be used as well as create global variables when specific variables have to be shared between files.

Behind The Scenes

The program can be split into two sections: user side and behind the scenes. The behind the scenes section includes the training data and chatbot model. Both of these processes don't affect the user directly while the user response creation does which is why they were split accordingly. Below are the pseudocode algorithms for the subroutines of this section.

MovieLinePairing

Initially, my plan for this was to find and pair up lines using only the “movie_lines.txt” file however, as I was doing research whilst making my pairing subroutine, I found that I could use the “movie_conversations.txt” file which already had all lines sorted into conversations and the order that they were said. This subroutine pairs up parent and child lines together. Because these are all pre-paired up, this saves me the time and processing effort of having to sort the movie lines together and then extract the data (whereas now I can simply extract the data).

```
FUNCTION MovieLinePairing(row) #this takes in the current row read in the movie lines text file
    new_list = row.split(" +++$+++ ") #creates a new list which consists of all of the IDs
    lines = new_list[position] #gets the IDs of all of the lines said in the "new_list"
    IF length(lines) != 1 #the length of lines is the length of the conversation
        ids = lines.split(',') #new array of line ids
        FOR line_id in ids: # "line_id" is now the id of one of the lines
            FOR data in movie_lines: # "movie_lines" is a list of all rows from
                # "movie_lines.txt"
                    child = data[position] #the movie line for the child
                    parent = data[position-1] #the parent is the line before
                    FileInsert(parent,child) #calls another subroutine
    END IF
END FUNCTION
```

I can call this function by running MovieLinePairing(row) where row is a row from the “movie_conversations.txt”.

FileInsert

This subroutine is fairly short and simple, all it's doing is inserting ('dumping') data into the JSON file to be stored and later extracted. There are two ways in which I can approach this subroutine: I can insert each paired data one-by-one as soon as they are paired together into the file or I can use a list to hold the paired data and do a mass insertion every x pairs formed (e.g. insert into the file every 200 pairs at once). The first idea is much simpler to make but the second idea could be more efficient and make this process faster as I am inserting more pairs at the same time thus opening and dumping into a file less times.

```
FUNCTION FileInsert(parent,child)
    counter = counter + 1
    data['intents'].append({"tag": counter, "parent" : parent, "child" : child}) #data
variable #is created outside of the function
    IF length(data['intents']) >= 200 #this is an example for if there are 200 pairs
        file.dump(data)
        data['intents'] = [] #refreshes the list
    END IF
END FUNCTION
```

Data is a global variable which is a dictionary stored like this {intents: []}, where the array will store a dictionary of parent and child messages. Counter is a global variable which starts at 0. This counter is used for tagging. Tagging will be used for assigning number categories to a set of parent and child messages which can be used by the chatbot model to predict which category was said by the user. This means that categories must be unique and since there will be hundreds of thousands of paired messages, the best way to ensure a unique tag is by using an incrementing number tag.

These are all of the subroutines for creating my training data. The next set of subroutines are the model subroutines which consist of converting the training data into data suitable to be used by a neural network and then training that neural network to recognise context.

Creating & Training a Model

This subroutine is the hardest subroutine in the program. Here, I will be creating a model using Tensorflow and, using my training data, I will be training the model to recognise certain patterns (using TFlearn). Before the model can be created, there would need to be a subroutine which converts the training data into numerical data. This will output data that the neural network will recognise and work with.

Generating numerical data:

```

FUNCTION NumericalData(documents, words, classes)
    FOR doc in documents
        bag = [] #the bag of words
        training = []
        parents = doc[0]
        FOR w in words:
            IF w in parents
                bag.append(1) #the word exists here
            END IF
            ELSE
                bag.append(0) #the word doesn't exist here
            END ELSE
            output_row = [0] * len(classes)
            output_row[classes.index(doc[1])] = 1 #gets the index of the category and
#turns the 0 at that position into a 1, showing which class the word is
            training.append([bag, output_row])
        training = np.array(training) #numpy array so the data structure suits Tensorflow
        train_x = training[:,0] #splitting the list at index 0, these are all of the bag of words
        train_y = training[:,1] #splitting at index 1, these are all of the output rows (shows
the
#positioning of data using 0's and 1's)
    END FUNCTION

```

In this subroutine, three variables are inserted: documents, words and classes. Documents are the categories of responses along with that response. For example, “Hello” would go under the greeting category so the documents list would store this as “documents[“Hello”, “Greeting”]”. Thus, the documents variable acts as a corpus. Words is the list of all words from all parent messages in the training data JSON file. Classes is a list of all the categories of messages in the training data. The two variables created here, “train_x” and “train_y” will be used in the next subroutine to create and apply training to the model.

Creating the model:

```

FUNCTION Model(train_x, train_y)
    network = tflearn.input_data(shape=[None, len(train_x[0])]) #build the neural
network
    network = tflearn.fully_connected(network, len(train_y[0]))
    network = tflearn.regression(network)
    model = tflearn.DNN(network, tensorboard)
    model.fit(train_x, train_y, n_epoch=1000, batch_size=8) #some standard settings
    model.save('model.tflearn') #saves the model after training
END FUNCTION

```

The value inserted into this subroutine are the two we created in our previous subroutine. There

were a lot of settings for tflearn, I will have to start with the default/recommended settings and later play around with the settings to find which configuration is most suitable for my model. However, the model has been configured to take in the numerical data that we have created in total.

User Side

The user side section consists of the user response creation. This process involves the user and is what the user sees happening on their screen. The user response creation creates a response using the model (this model will use machine learning to determine the best possible response) which will be represented on the user interface.

Bag Of Words

This subroutine will be used to structure the user's message into numerical data, similarly to how I structured the training data. This numerical data will be held in a bag of words. This subroutine would be used for finding the context of a message and constructing a suitable response.

```
FUNCTION BagOfWords(sentence, words)
    bag = [0] * len(words) #first creates the empty bag
    FOR word in sentence
        FOR i,w in enumerate(words)
            IF w == word
                bag[i] = 1 #where the word is located in the bag,
                there will #be a 1
            END IF
        return(np.array(bag)) #returns a numpy array of the bag of words
    END FUNCTION
```

The two inputs here are “sentence” and “words”. Sentence is a string variable containing the sentence/words that need to be turned into a bag of words. Words is a global variable containing all of the words used in the parent messages in the training data.

Classifying User Responses

I have created subroutines before which created a chatbot model. Now I need to use my chatbot model to classify which parent message the user input is most similar to. This will be used in the next subroutine to find the context of the message and form a response.

```
FUNCTION Classify(sentence)
    bag = BagOfWords(sentence, words) #turns the user input to a bag of words
    results = model.predict([bag]) #calculates the similarity of the user input
```

```
#to all parent messages in the database
    FOR i,r in enumerate(results)
        IF r > 0.25 #this value might change to increase how similar the message
#must be to pass this
        results = [i,r]
    END IF
    results.sort(key = lambda x: x[1]) #sorts by strength of probability
    class = []
    FOR r in results
        class.append((classes[r[0]], r[1])) #a tuple that contains the class and the
#probability
    RETURN class
END FUNCTION
```

The input of this subroutine is the variable “sentence” Sentence is the user input/message. Classes is a global variable that has been defined outside of the subroutine and is the list of all classes (tags) in the database (the training data), this means that all pairs in the database must be uniquely assigned a tag. Words is another global variable which is a list of all of the words stored (ordered alphabetically) in the parent data of the training database. In this subroutine, I return a class which is a category of parent and child message.

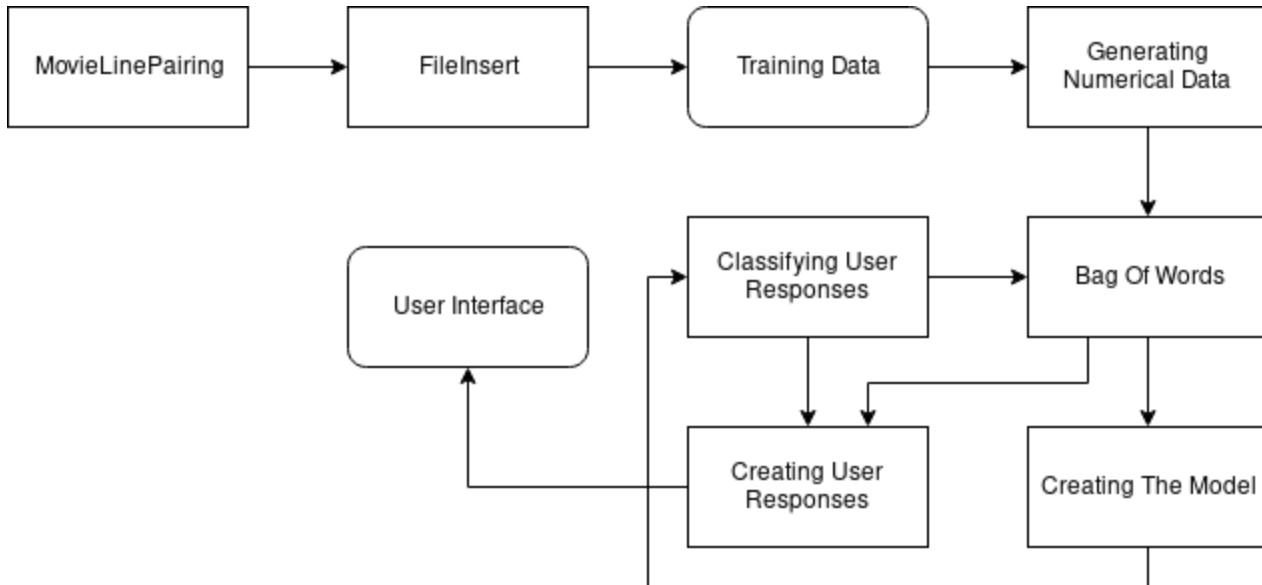
Creating User Responses

This subroutine is very important for the machine learning aspect of the bot. It will use the two subroutines above, bag of words and classifying user responses, to find the context of the user’s input and form a response according to the most similar message to the context in the training data (the database). This response will then be sent to the user interface module where it will be displayed to the user. Do note that although the name of this subroutine is “Creating User Responses”, the user’s response isn’t being created but in fact the chatbot’s response to the user.

```
FUNCTION Response(sentence)
    results = Classify(sentence)
    IF results != null #if there are possible predictions for the message
        WHILE results != null
            FOR i in intents #i is a set of paired messages + their tag
                IF i['tag'] = results[0][0] #finding a tag matching the
first #result. This will work each time as after each index, the first one is popped
                responses = [i['child']]
                RETURN responses
            results.pop(0)
        END IF
    END FUNCTION
```

There is only one input in this subroutine which is the variable “sentence”. This variable represents the user’s message. The returned responses will give a list of possible responses to the sentence variable according to the prediction of the model.

Here is a diagram showing how all of these subroutines link together:



Testing Algorithms & Trace Tables

Most of the algorithms can be tested using trace tables except any algorithms involving the models (especially those involving Tensorflow or TFlearn). This is because the neural network isn’t like normal programming, it uses strings to convert to binary integers (1’s and 0’s) so it can’t be tested so simply. In order to test this, I can create test training data to form a test training model. Then, I can run a series of tests to see if the model is correctly identifying the context of my message and if the bot’s response is suitable to my message. This can only be done during development which will increase the testing workload.

I could use the following template trace tables for each algorithm:

Line	Variable1	Variable2	Output

Trace table for MovieLinePairing:

Example of a row: row = “u0 +++\$+++ u2 +++\$+++ m0 +++\$+++ [‘L194’, ‘L195’, ‘L196’, ‘L197’]”

Line	Iteration	New_list	IDs	Line_id	Parent	Child
2	-	[u0, u2, m0, ['L194', 'L195', 'L196', 'L197']]	-	-	-	-
5	-	[u0, u2, m0, ['L194', 'L195', 'L196', 'L197']]	['L194', 'L195', 'L196', 'L197']	-	-	-
6	1	[u0, u2, m0, ['L194', 'L195', 'L196', 'L197']]	['L194', 'L195', 'L196', 'L197']	'L194'	-	-
6	2	[u0, u2, m0, ['L194', 'L195', 'L196', 'L197']]	['L194', 'L195', 'L196', 'L197']	'L195'	-	-
6	3	[u0, u2, m0, ['L194', 'L195', 'L196', 'L197']]	['L194', 'L195', 'L196', 'L197']	'L196'	-	-
6	4	[u0, u2, m0, ['L194', 'L195', 'L196', 'L197']]	['L194', 'L195', 'L196', 'L197']	'L197'	-	-
8	1	[u0, u2,	['L194',	'L194'	-	"Can we

		m0, ['L194', 'L195', 'L196', 'L197']]	'L195', 'L196', 'L197']			make this quick? Roxanne Korrine and Andrew Barrett are having an incredibly horrendous public break- up on the quad. Again."
8	2	[u0, u2, m0, ['L194', 'L195', 'L196', 'L197']]	['L194', 'L195', 'L196', 'L197']	'L195'	"Can we make this quick? Roxanne Korrine and Andrew Barrett are having an incredibly horrendous public break- up on the quad. Again."	"Well, I thought we'd start with pronunciation, if that's okay with you."
8	3	[u0, u2, m0, ['L194', 'L195', 'L196', 'L197']]	['L194', 'L195', 'L196', 'L197']	'L196'	"Well, I thought we'd start with pronunciation, if that's okay with you."	"Not the hacking and gagging and spitting part. Please."
8	4	[u0, u2, m0, ['L194',	['L194', 'L195', 'L196',	'L197'	"Not the hacking and	"Okay... then how 'bout we try

		‘L195’, ‘L196’, ‘L197’]]	‘L197’]		gagging and spitting part. Please.”	out some French cuisine. Saturday? Night?”
--	--	--------------------------------	---------	--	--	--

This trace table shows that the algorithm for pairing lines together works.

Trace table for FileInsert:

Using parent = “Well, I thought we'd start with pronunciation, if that's okay with you.“ and child = “Not the hacking and gagging and spitting part. Please.”

Line	Parent	Child	Data
1	“Well, I thought we'd start with pronunciation, if that's okay with you.“	“Not the hacking and gagging and spitting part. Please.”	-
2	“Well, I thought we'd start with pronunciation, if that's okay with you.“	“Not the hacking and gagging and spitting part. Please.”	{intents: []}
3	“Well, I thought we'd start with pronunciation, if that's okay with you.“	“Not the hacking and gagging and spitting part. Please.”	{intents: [{parent: “Well, I thought we'd start with pronunciation, if that's okay with you.“, child: “Not the hacking and gagging and spitting part. Please.”}]}{

With a trace table I can see that I am sending in the correct general format of the data. With this algorithm I can't test whether the data being dumped is being dumped correctly into the file or stored correctly in the file until after this whole process has been completed. This means that I will have to spend time in the development section to test if the training data has been stored correctly.

Creating a trace table for the two algorithms: “Generating Numerical Data” and “Bag Of Words” would not be possible because I can't predict where the 0's and 1's will be located before finishing my training data and having a list of all parent words created. Without knowing what

this list contains and where the words would be located, I can't determine where the 1's will be thus I will have to test if the numerical data has been structured correctly during the development stage.

Explanation & Justification Of This Process

The initial task seemed large and complicated but by breaking it down into separate parts, it makes the development of this project much easier. This way, I can focus on the functions of separate sections and how they individually work before combining them together to make up the whole project.

I have split my program into two parts: the User Side and the Behind The Scenes. This split was made here because there are parts of the program which are involved with creating a model and how it works which is mostly only done once. The result of these subroutines provide the program with capabilities or data for the User Side. For example, the training data created in the Behind The Scenes is used in two ways. Firstly, it's used to train the chatbot model which will be used in the User Side to create a prediction to what the user message is. Secondly, after the model makes a prediction, I need to form a response for the chatbot which will use the training data. The User Side handles the interactions with the user such as the user interface and (as mentioned in the example above) response creation.

An earlier diagram above showed a basic idea of how the subroutines link together. MovieLinePairing pairs up parent and child messages in movie scripts. This data need to be stored in a file database so this data is sent off to the subroutine FileInsert. FileInsert will create this file and add the data to it. A complete file forms the training data. These two subroutines therefore make up the process of creating my chatbot's training data. The subroutine Generating Numerical Data uses the training data to convert all of the parent messages to numerical data which is suitable for use in neural networks. Parts of this numerical data is then used by the subroutine Bag Of Words to create a bag of words (which is one of the products of the numerical data subroutine which will be used for training patterns of words). Another subroutine which links to the Bag Of Words subroutine is Classifying User Responses. A user response will be a string message. However, neural networks are not able to understand strings therefore they have to be converted to a bag of words first. The training data bag of words is used by the Creating The Model subroutine to create a model which is configured to take in the same data type (the same length of bag of words each time). The model will be trained using the bag of words and another numerical data. The trained model will use the Classifying User Responses subroutine and Creating User Response (where the user response has been converted to a bag of words) to create a response. The classification and creation of user responses is a part of the User Side. The response created by these subroutines will be sent off to the user interface and displayed to the user.

Breaking down the problem like this ensures that my application is in separate modules that all communicate with each other. If there's a problem with the program, it's easier to identify the origin and fix it because of the separation as well as isolating the problem. Fixing problems on separate modules ensures that the entire program stays intact so if a module breaks down, the

other modules are not affected.

The code should stay very similar to the subroutines unless I decide to make any changes in the development section. I will achieve this by following the design and subdividing each part of the code into separate modules. Inside these modules, the code will be divided into separate functions. If another developer were to make changes to my code, it would be very easy as each section has been separated according to function, for example the creation of training data and model training are in two separate sections which do not directly affect each other. This means that if a developer wanted to change the way in which the model was trained, the training data (and all the other modules) would not be affected and they could simply go to the training section to complete these changes. Each module of the program will need to communicate with each other in some way so I will need to create global variables to pass on information for example creating a chatbot message string in the response creation module which can be accessed by the user interface module. This makes the code simpler because in most cases, I would not need to use functions with return statements to pass on information.

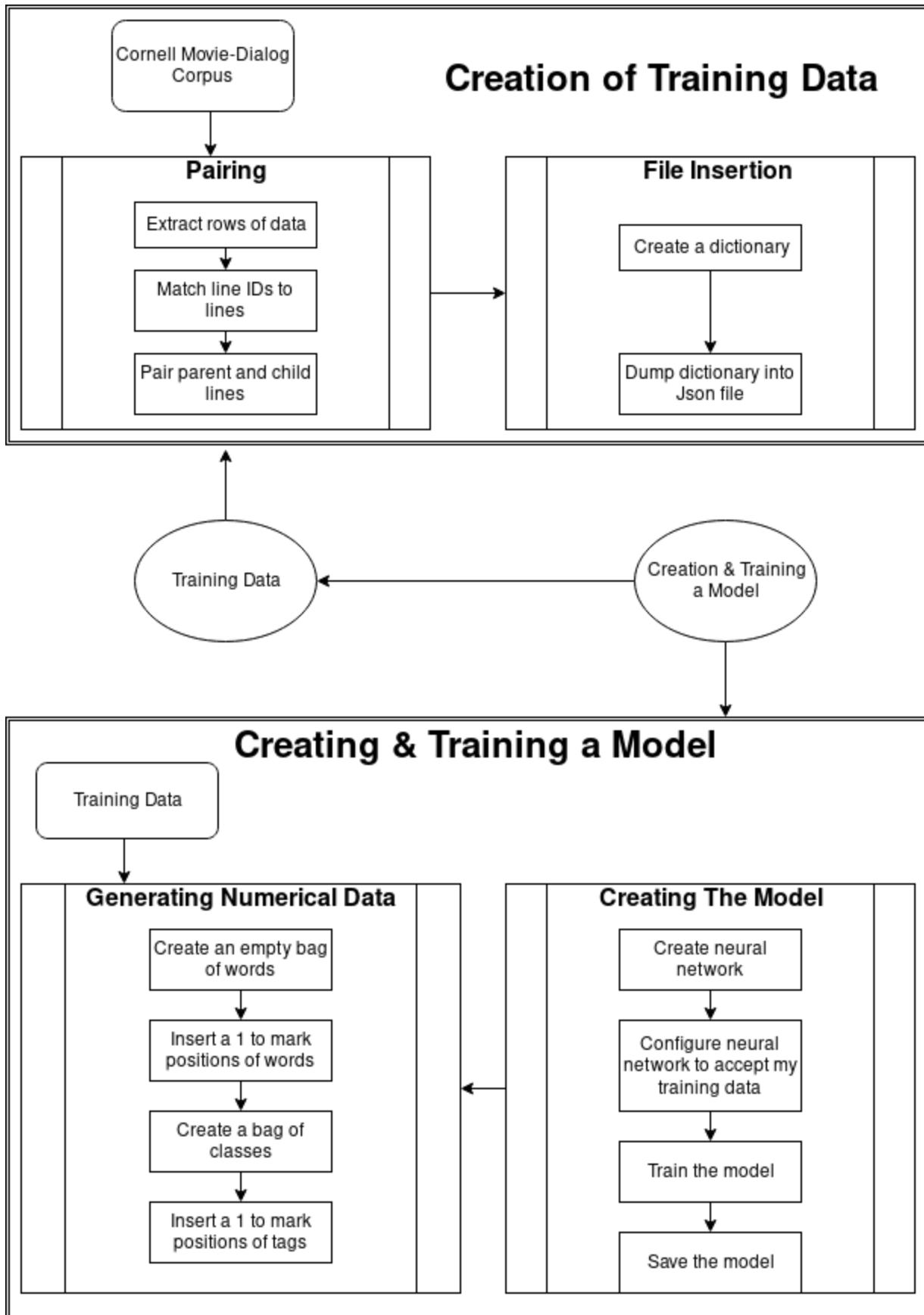
The greatest benefit of the design is that the computational thinking used allowed me to plan my program before reaching the development stage. This breakdown ensures that I have thought about how the algorithms in the program works and how they all link together to form the chatbot application. This planning decreases the amount of testing required for the development stage as I am more aware of what each algorithm needs to do therefore I don't need to play around with code as much to see if the output is the required result.

Links to success criteria:

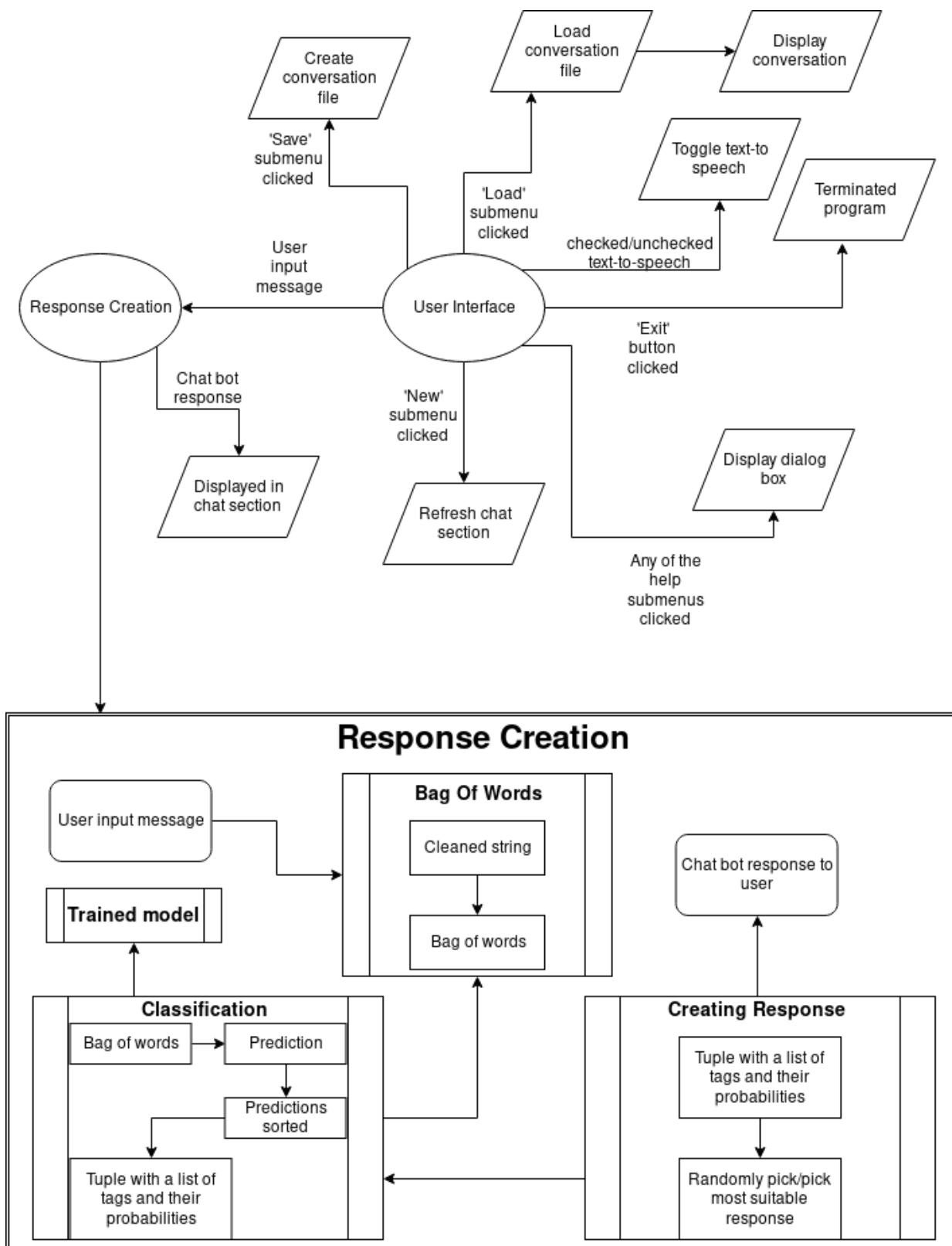
- Training data creation
- Training of the chatbot
- The bot is able to respond with a relevant reply
- Use of machine learning for responses

Diagrams To Show How The Subroutines Link

Behind The Scenes



User Side



Inputs And Outputs

Input	Process	Output
User message via enter key.	Message decomposition algorithm (removing unnecessary parts of the message) and the bag of words algorithm.	User's message as a text message displayed in the user interface (the chat section).
Bag of words.	User response creation algorithm.	A response to the user's message (the chatbot's message) is displayed in the user interface.
Exit button.	The program will stop and terminate.	The chat conversation has stopped.
Loading conversation file.	Wipes the chat section clean and loads the conversation file in the same format as stored.	New conversation inserted to the chat section and displayed.
Save as a file.	Extracts all of the strings in the chat section and keeps the same formatting.	Text file containing all of the conversation data.

Key Variables

Name	Data Type	How It Is Used
intents	A 1D array which stores a large dictionary. There are only strings stored in this dictionary.	The dictionary contains the training data. There's the parent, child and tag of each pair of messages stored here.
punc_dic	A small dictionary.	This dictionary contains all punctuations which are to be removed from a message string so it can be "cleaned". E.g. a comma.
words	A large 1D array storing only strings.	The words list contains all of the unique words used in the parent messages in the training data.
classes	A large 1D array storing only strings.	The classes list stores all of the tags in the training data. Even though tags are numerical, they are stored as strings in the JSON file.
documents	A large 2D array.	Documents stores the list of the tokenized words from the parent message as well as the class it's in together, bracketed.
train_x	A large 2D array storing lists of combinations of 0's and 1's.	Train_x is storing a list of all of our bag of words from the parent messages from the training data.
train_y	A large 2D array storing lists of combinations of 0's and 1's.	Train_y stores a list of the location of the class that a parent message is in.
reply	A string which is a global variable.	This string holds the chat bot's response which is then

		used in the user interface to display it to the user as a bot message.
--	--	--

Validation

To make sure that the application is robust, all of the user inputs must be validated to make sure that they are inputting the correct data. These are the two types of data that the user can input and how they will be validated:

Buttons

The exit button that I will be using is part of Tkinter so it is already robust. By nature of how this button works, if it is pressed when it isn't linked to anything, nothing will happen. When it is pressed and linked to a function, it can't pass any variables to the function so there is no risk of sending incorrect data.

Menus & Sub Menus

I will be using several menus and sub menus to allow the user to change the settings of the program and save/load their conversations with the chatbot. Menus and sub menus are a part of Tkinter and each sub menu links to a function. If it is pressed when it's not linked to a function, nothing will happen. When it is pressed and linked to a function, it doesn't itself pass any variables to the function so like the buttons, there's no risk of the menu sending any incorrect data.

Text Boxes

Text boxes are different to buttons as the user can enter any text that they want. The text box in this application will be the message input, where the user can write a message to the chat bot. The validation that I will have to perform would be to test that the input is a string and that it is a valid key on the keyboard (for example, alphanumeric characters). I would have to check that it's always lower case and if it's not, that it is changed to the correct case. By checking the input before the application uses it, I can ensure that it will not cause any issues during the message analysis and response creation processes.

Testing Method

Throughout development, each function should be tested as it's created to make sure that it works. This means that the program will function when complete and then any further testing will highlight bugs or feature improvements.

The testing during the development should include the data input to the program (which mainly is the message that the user has entered) and the result with it being the chat bot's response. I can check that the result is what was intended by checking the user interface and seeing if there has been a correct bot response displayed. The application console output should be shown in case of errors or any crashes. As well as this, for debug purposes, processes during the user message analysis will be printed as well as steps in the process of the program deciding what the bot's response is.

After the development is complete, a final test should be carried out by myself which will include destructive testing and ensuring that all intended features are there.

Destructive testing will be entering incorrect data and seeing how the program handles it. Since it's not possible for me to enter incorrect data on an exit button, this will be tested on the user input and the loading of files. For the user input, I will try a variety of incorrect entries to see what happens, for example, use of unknown characters. For the loading of files, I will create text files that are structured irregularly and different from the main structure of a normal conversation file to see how the program handles it.

I will record any data that I input and will show screenshots of its result on both the user interface and the debug in the program's console.

After the testing and any final fixes, I will show the program to the stakeholders to ensure that it meets their requirements and to see if they have anything they would like added or amended.

Iterative Development

I will be using iterative development to complete this application. Using iterative development means that I will be breaking the program down into smaller, solvable problems and then solving them in an iterative process (all of the smaller problems have been explained in the design section). This means that after each stage is completed, it will be tested to gradually create a prototype of the program. This process repeats until the program is complete.

Using this method of testing will help with this process as it shows how I will test the program as it is being developed and how I will complete a final test.

Text Inputs

For the main part of the application, the input will be from the input box. Due to the input being text, I can prepare strings to use for testing.

For the text analysis, I could use a variation of sentences consisting of a selection of very vague to very direct sentences. I could also use different variations of message length for example some string input could be sentences whilst others can be paragraphs. This way, I can check that my program is able to respond to different types of messages.

An example string that I can use is “The sky is nice.”. This is a vague sentence because it does not require the chatbot to respond in a particular way. While it could reply to the statement with “I agree!” or “Sure.”, it could completely deflect the sentence by asking the user a question. However, it would be better for me to use a specific sentence like “What’s your favourite colour?” as this would create very consistent and predictable results. With a vague message, you cannot predict what the chatbot may respond with as it could be completely random but with a specific message, you know that the response has to be related to, like in the example, colours.

All of the text analysis and response creation algorithm troubleshootings will be done by using text inputs in the input box. I will document the messages that I entered as well as the responses given by the user by providing screenshots showing the chat section in the program.

Testing Checklist

The following is a checklist that I will be using to make sure that I have tested all of the functionality of the program.

What Is Being Tested?	Does It Work?
Check all text for formatting errors	
Exit button closes the application	
The user can’t modify the chat section	
Scroll bar works to see message history	
Saving a conversation into a file	
Loading a conversation file	

Creating a new conversation in the same session	
User is able to enter messages in chat	
Bot can respond to messages in chat	
Bot can respond using machine learning	

When this comprehensive list is complete, I will be sure that all the functionality of the program is met and that there are no errors. This means that the program can be passed onto stakeholders for their final input. At this stage, all the criteria on the success criteria will be met and the development complete.

Interface Inputs And Test Tables

There will be a text input box to let the user type any keyboard character. These will form the messages that the program will analyse and create a response to. Because the chat bot is able to read words, any letters, punctuation symbols and message characters (such as £) will be read and removed in the analysis. Any other characters will have no meaning to the text therefore will be incorrect inputs. To test the inputs, I will prepare a series of correct and incorrect inputs to make sure that the program validates properly:

Input	Valid/Invalid	Result
A-Z alphabet (in capitals)	Valid	
a-z alphabet (in lowercase)	Valid	
Word	Valid	
She said, "Hello!"	Valid	
£10	Valid	
*	Invalid	
@	Invalid	
	invalid	

The parts of the success criteria that this testing will satisfy are:

- Text area for input

Buttons

The functionality of the buttons is managed by Tkinter so they should be working without crashing. However, the action they perform is programmed by myself so I will need to check that they function in the correct way. This will be a part of the final testing and it will involve pressing each button in the program. Since I only have one button in the program, I will only have to test this one and ensure that it is doing what it is intended to do.

The parts of the success criteria that this testing will satisfy are:

- Option to press a button to stop the program

Sub Menus

Menus are implemented as widgets, just like the buttons and user input (the entry). In the user interface, these menus will open their own cascading menus, otherwise called sub menus. This means that the menus will also be managed by Tkinter. This also means that for any menus which display e.g. an information box, it would also be managed by Tkinter. However, some sub menus have added commands to them for example (taking the suggestion from my stakeholder, Gabriella) saving a file and loading a file. I would have to conduct a series of tests to see if the sub menus work as intended by using them normally and purposely trying to create errors to see how my program handles them.

Development & Testing

Stage 1: Training Data Creation

My program will be split into several sections with it totally consisting of four modules: Training Data Creation, Training, Response Creation and User Interface. In this section, we will be focusing on the Training Data Creation.

In order to create a machine learning chatbot, I will need to spend some time on creating my bot model and teaching that model how to speak. In order to teach my chatbot anything, I will have to gather some sort of database which contains a message (the parent) and its reply (the child). I already decided earlier in the project that I will be using the Cornell Movie-Dialog Corpus to gather my data as this corpus contains 200,000+ conversations from movies. Getting data from a movie script is great for what I need because there are so many different types of conversations happening (which allows me to create a more diverse model) and the line in a scene is linked to the next one/the one before.

Before I start coding, I need to look at the Cornell Movie-Dialog Corpus properly to see how the data is stored. This is done so that I know how to extract the data. Below are extracts from the files and how I intend on using them.

movie_characters_metadata.txt:

```
u0 +++$+++ BIANCA +++$+++ m0 +++$+++ 10 things i hate about you +++$+++ f +++$+++ 4
u1 +++$+++ BRUCE +++$+++ m0 +++$+++ 10 things i hate about you +++$+++ ? +++$+++ ?
u2 +++$+++ CAMERON +++$+++ m0 +++$+++ 10 things i hate about you +++$+++ m +++$+++ 3
u3 +++$+++ CHASTITY +++$+++ m0 +++$+++ 10 things i hate about you +++$+++ ? +++$+++ ?
u4 +++$+++ JOEY +++$+++ m0 +++$+++ 10 things i hate about you +++$+++ m +++$+++ 6
u5 +++$+++ KAT +++$+++ m0 +++$+++ 10 things i hate about you +++$+++ f +++$+++ 2
```

I can see that this file stores information about every single character in the movies. This data includes a unique character ID, their name, the unique movie ID (of the movie that they appear in), the name of the movie, their gender and a final value which I am not sure of. All of these are separated with the “+++\$+++” tag for easy extraction. My program doesn’t care which character said what and where it came from so this file can be ignored.

Movie_conversations.txt:

```
u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L194', 'L195', 'L196', 'L197']
u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L198', 'L199']
u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L200', 'L201', 'L202', 'L203']
u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L204', 'L205', 'L206']
u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L207', 'L208']
u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L271', 'L272', 'L273', 'L274', 'L275']
```

This file stores information about movie conversations (hence its name). Each line consists of: two character IDs (the character who said those lines in the conversation), the ID of the movie which this conversation occurs and a list of the IDs of all the lines said in this conversation. All of the movie lines are sorted in this file by conversation which means that I don't have to bother with coding my own line sorting system.

Movie_lines.txt:

```
L1045 +++$+++ u0 +++$+++ m0 +++$+++ BIANCA +++$+++ They do not!
L1044 +++$+++ u2 +++$+++ m0 +++$+++ CAMERON +++$+++ They do to!
L985 +++$+++ u0 +++$+++ m0 +++$+++ BIANCA +++$+++ I hope so.
L984 +++$+++ u2 +++$+++ m0 +++$+++ CAMERON +++$+++ She okay?
L925 +++$+++ u0 +++$+++ m0 +++$+++ BIANCA +++$+++ Let's go.
L924 +++$+++ u2 +++$+++ m0 +++$+++ CAMERON +++$+++ Wow
L872 +++$+++ u0 +++$+++ m0 +++$+++ BIANCA +++$+++ Okay -- you're gonna need to learn how to lie.
```

This file stores data regarding the lines said in the movies which are the unique line ID, the ID of the character who said this line, the name of the character and the line said. This file is important because it contains the actual data that I want to extract, the lines said. Because both the "movie_conversations.txt" file and this file store unique line IDs, I can use this variable to link both the files together. For example: I can look at the first conversation and look for the line that corresponds to the first ID to determine what was said. This acts like a relational database where the line ID is a primary key in the "movie_lines.txt" file but acts like a foreign key in the "movie_conversations.txt" file, allowing a link.

Movie_titles_metadata.txt:

```
m0 +++$+++ 10 things i hate about you +++$+++ 1999 +++$+++ 6.90 +++$+++ 62847 +++$+++ ['comedy', 'romance']
```

This file stores information about the movies in this corpus such as its unique ID, its name, the year it was made and the genre it fits into. This data is completely useless to my program because it doesn't directly relate to a parent or child message.

All of this research was done before the development of the project, in the design section so that I could plan how my project was going to work.

Before designing my subroutines for the training data, I came up with a basic idea of how I want the program to work which can be shown in the code below. This idea of uses “movie_lines.txt” to sort out and pair up movie lines. Even though the subroutines are different to this idea, I can use this to code the basics of the program before creating the subroutines.

```

linked_list = [] #A list which will contain the parent and child message

def dataFormatting(data): #A Function for formatting the data to be inserted
    new_list = data.split("+++$+++") #Creates a new list with all of the data in a row.
    The list looks like this [conversation ID, character ID, movie ID, character's name, message]. All of the data is separated by the tag "+++$+++"
    if len(new_list) != 5: #Looking at the "movie_lines.txt" file, each row has 5 pieces of data separated with 5 tags. If this row (by chance or corruption) is not like the other rows, I don't want to use the data from this row
        return #Therefore, I return, ignoring this row and proceed to the next
    message = new_list[4] #The line said (message) is the 5th item of the list
    movie_id = new_list[2] #The ID of the movie is the 3rd item of the list
    #print(message) #Printing the values for debugging to see if they are correct according to their value in the file
    #print(movie_id)
    global linked_list #Creates a global variable so that this function is able to access and modify the parent-child list
    if len(linked_list) == 2: #If the list is storing two messages, the parent and child messages, it implies that it has them linked together in the list (because the file is sequential). That means that I can run a couple of checks to ensure that these are in fact parent and child messages and then insert them together into my Json file
        #TODO check if the lines are said in the same conversation
        #TODO check if it is the end of the movie so that messages between movies aren't linked
        #TODO add linked messages to a json file
        linked_list = [] #After the message has been inserted into the Json file, I can reset the list to store the next linked pair
    else: #If there aren't two messages stored here
        linked_list.append(message) #Add a new message
    #print(new_list, '\n') For debugging, I can check if the pairing has been correct according to what the lines were (for example, if the correct two lines were taken from the file and paired together)

with open('movie_lines.txt', 'r', encoding = 'iso-8859-1') as f: #This line allows me to open my file, without using a file.close() line of code at the end so this file will only remain open to be read when it's being used. The default encoding (UTF-8) can't be used, an error is displayed because some characters in the file are not supported.
    for row in f.readlines(): #Reads each row in the file
        dataFormatting(row) #The data is not formatted properly to be used for training so it must be altered

```

This code shows the beginning part of the process of linking up two messages. However, when comparing this code to my training data subroutine, I can see that the subroutine does this most efficiently as it takes advantage of the fact that the data has been sorted for me. This code is an outline of the main structure. I have noticed, however, a minor problem with this code. At the top, I am creating a list which will store the parent and child messages then I am defining the list as a global variable inside the function so that it can append values to it. The problem lies at the end of the function where, after the list has been inserted into a JSON file, I reset the list to be empty. If the list is going to end up empty for every single row, there's no point in me creating this list at the beginning and defining a global variable and later resetting. Instead, to do the same thing, I can define a new blank list at the beginning of the function which will reset for each row (because the function is called in the for loop below).

Furthermore, I am pairing up movie lines together from the “movie_lines.txt” file under the assumption that the next movie line (if in the same movie and conversation) is the response to the current line. However, this may not be the case depending on how the movies have been split up into this file. Therefore, some pairings may not be correct.

Further explanation of what the code is doing:

I want to link up parent and child messages so I created a global list called “linked_list” so that this can be accessed by the “dataFormatting” function and any other functions that I will be adding later. When I looked at the “movie_lines.txt” file earlier, I could see that each important variable in a line is separated by the tag “+++\$+++”. Because of this, I can create a new list which contains these variables by splitting rows from the file by this tag. This new list will then contain all of the ID's, character names and lines in the following format [conversation ID, character ID, movie ID, character's name, line]. In this way, I can gather all of the information available about lines said in all of the movies stored in the corpus.

Each new list should always have exactly 5 variables however I check this incase there are any errors in the file. If the list doesn't contain 5 variables, I know that this particular line is not formatted correctly thus it may lead to errors when using so to prevent use, the program simply returns and continues on with the next line. This is important because I don't want my chat bot to learn a parent and child message that aren't actually the parent/the child of the other.

To open any of the files from the Cornell Movie-Dialog Corpus, more specifically “movie_lines.txt”, I have to use a specific encoder. This is because the corpus contains English movies from all around the world which results in characters which aren't supported by UTF-8 (the default encoder) in the text file. However, ISO-8859-1 does support all of the characters in

the text file thus it must be used instead of UTF-8. Using UTF-8 encoding to open the file and then reading it results in the following error:

```
File "/home/danielle/Documents/untitled1.py", line 10, in <module>
    for r in file.readlines():

File "/usr/lib64/python3.7/codecs.py", line 322, in decode
    (result, consumed) = self._buffer_decode(data, self.errors, final)

UnicodeDecodeError: 'utf-8' codec can't decode byte 0xad in position 3767: invalid start byte
```

The parts here that I have left to do I marked with comment “#TODO” tags along with what needs to be done. My initial plan is to store the line of the current row and then somehow find the line of the next row of the file. Then, I would need to check if it’s part of the same conversation in the same movie (whoever, considering the conversation tags are all unique, I can simply compare only those). If the next line is in the same conversation and movie, I can say that this is a response (child) message to my current (parent) line. Therefore, I can store both of these in a JSON file together.

Looking back at the research I have done earlier and the subroutine created, it is clear to see that the subroutine designed the better version. It gives me a much easier way of linking movie lines together as they are already stored together in conversational categories. This would be because of the file “movie_conversations.txt”. This stores data in the following way:

```
u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L194', 'L195', 'L196', 'L197']
u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L198', 'L199']
u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L200', 'L201', 'L202', 'L203']
u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L204', 'L205', 'L206']
u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L207', 'L208']
u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L271', 'L272', 'L273', 'L274', 'L275']
u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L276', 'L277']
u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L280', 'L281']
u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L363', 'L364']
u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L365', 'L366']
u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L367', 'L368']
u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L401', 'L402', 'L403']
u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L404', 'L405', 'L406', 'L407']
```

This is a list of all of the conversations in all of the movies in the corpus. This is stored in the following format, separated by tags: first character’s ID, second character’s ID, movie ID, [ID of lines (in the order that they were said)]. This is the perfect solution for me because not only can I guarantee that the parent and child message come from the same movie and conversation, I can also ensure that the parent line is actually the parent of the child. Because I will only be receiving the parent and child ID’s, I will still need to use “movie_lines.txt” to find out what the

actual lines said are. This is much easier than the first concept of this extraction, where I would have to create my own pairing process.

The code for my new plan can be seen below:

```
def pairing(row): #The pairing function, used to be called dataFormatting
    conversation_texts = [] #This array will store all of the strings said in a conversation
    position = -1 #This variable marks the index of the current line when using the 2nd for loop below. It starts at -1 because the variable
    will increment until it's the last value so at -1, the first index will be 0
    new_list = row.split(' +++$+++ ') #This new list consists of: [character1, character2, movieID, [lineID, lineID, ...]] the line IDs vary
    according to how many lines have been said in this conversation
    if len(new_list) > 4: #The list has to have 4 pieces of data only, the 2 characters who are speaking to each other, the movie which this
    conversation is from and a list of the lines exchanged in this conversation
        return #If the current row isn't stored in the correct format e.g. an extra value because of corruption inside the file, I don't want
        to use this conversation
    lines = new_list[3] #The lines variable holds the list of line IDs from the conversation
    if len(lines) == 1: #If there is only 1 line said in this conversation, there's no pairing so we don't want this line is not a parent or a
    child
        return #So we move on to the next row in the file
    else: #If the conversation does have at least a parent and a child
        line_ids = lines.split(',') #The line ids are formatted like a list but it's actually a string so it needs conversion into a list so I
        can iterate through all of the IDs
        for line_id in line_ids: #For each ID in the list of line IDs said in this conversation
            position += 1 #The position value increments to mark the index of the line ID. This will allow me to determine if the line is a
            parent or a child
            formatted_id = line_id.replace("[", "").replace("]", "").replace("'", "").replace(" ", "") #The ids are compromised by array
            symbols (because it was an array stored as a string) so they must be removed in order to compare IDs in "movie_lines.txt"
            for line_data in movie_lines: #Searching through the movie lines corpus
                if formatted_id == line_data[0]: #Check to see if the IDs match
                    if position == 0: #Check if this is the first id in the list this doesn't have a parent but it is the parent of the next
                    message
                        conversation_texts.append(line_data[4].replace("\n", "")) #Appends the first message, it's the first parent so I will
                        need this data for the 2nd message which is the child
                    elif position == len(line_ids)-1: #If this is the last id in the list this can't be the parent of another message it can
                    only be a child
                        child = line_data[4] #Setting this line as the child
                        parent = conversation_texts[position-1] #Finding the parent from the conversation (it's always the message before)
                        #fileInsertion(parent, child) Inserting into the file
                        #print("parent:" + parent + "child:" + child) For debugging
                        #print(conversation_texts)
                    else:
                        #A value that passes to this else statement is a value that isn't 1st or last which means that it acts as a child for
                        the value before and the parent of the next value
                        conversation_texts.append(line_data[4].replace("\n", "")) #As a parent for the next value, it must be appended so it
                        can be found by the next child
                        child = line_data[4].replace("\n", "") #Now it acts as the child
                        parent = conversation_texts[position-1] #The parent is the value before
                        #fileInsertion(parent, child) Inserting into the file
                        #print("parent:" + parent + "child:" + child) For debugging
                        #print(conversation_texts)

    with open('movie_lines.txt', 'r', encoding='iso-8859-1') as file:
        for r in file.readlines():
            temp_list = r.split(' +++$+++ ') #movie_lines.txt" also has its data stored separated by tags,
            similarly to the "movie_conversations.txt"
            movie_lines.append(temp_list) #2D array stored as follows - [index of row of line][which value you
            want e.g. 0 is line id]
```

Note: movie_lines is an empty array defined outside of this function - "movie_lines = []"

```
with open('movie_conversations.txt', 'r', encoding='iso-8859-1') as f:  
    for row in f.readlines(): #Reads all of the rows in the file, the value row equals the line in the file  
        corresponding to that row  
        pairing(row) #Calling the function to extract the data from this row
```

This code works to use the Cornell Movie-Dialog Corpus to create a database in a JSON file (stored as dictionaries). Firstly, I use “with open(...)...” twice in this code. Once for “movie_lines.txt” and another for “movie_conversations.txt” which will hold all of the outputs (the data extracted) from the corpus files. Using “with open” works the same as only using “open” to open the file and using another line of code which closes the file when it stops being used. This is helpful as I don’t need to write “file.close()” twice (for whenever I open the files). Therefore, the files will only be read when they are being used. Because I am going to need to use the movie lines to compare line IDs and find out the message sent in the line, I am going to have to store the rows in “movie_lines.txt” in a 2D array. This is done so that I can create my own corpus of extracted lines which I can use to find the equivalent message said from the line ID. From my previous version, I know that the line ID is stored in the first item of the array in the movie lines array so I can use a for loop to loop through all rows and compare IDs. When there is a matched ID, the second item represents the line so I can find the index of this match and retrieve the line. There is also a counter variable used called “position”. This is used to check the position of an ID in a conversation. For example, if the position = 0, I know that the ID is the first item in that array so it’s the first message said in that conversation. Using this is very important because I have 3 different types of lines:

1. A line that is a parent for a message but doesn’t have a parent itself (so it’s not a child). This is the first line as it will never have a parent (since it’s the first one said) but it will always be the parent of a child message (the 2nd message). This scenario will always occur because a conversation will always be at least two messages long (as the program returns when it isn’t).
2. A line that is both a parent and a child. The line before it is its parent and the line after is its child. This applies for lines said in the middle part of the conversation. This scenario won’t always occur, for example, a conversation which is only two messages long will not contain a middle message.
3. The final type is a line which is only a child. This applies to the last line because there’s no line after this for it to be a parent. This scenario will always occur because there will always be a final message said in a conversation.

I deal with this types of lines very differently. Firstly, I have to check what type of line it is. I know that type 1 is always the first message so I need to check if the position = 0 and apply the appropriate code there. I know that the last message is if the position is equal to the length of the conversation (amount of lines said) -1 (because position starts at 0). There are several

lengths of conversations (e.g. 3, 5, 12 messages said to each other) so I can easily use “len(line_ids)” to find how many lines there are. Even though the list “line_ids” is actually a list of line IDs and not the lines themselves, there will still be an ID per line in the conversation therefore, in this way, I can still find out how many lines were said in that conversation. I use an “else” statement for type 2 as type 2 can’t be type 1 or 3 at the same time so it will be anything else.

For type 1 (the first message), I know that it will only ever be a parent. This is an important line because this will link to the next message which will be its child. I will only be linking messages for type 2 and 3 so I have no use for type 1 other than to insert it into my conversation list (“conversation_texts”) to use for later. Note that all lines have to be formatted before inserted into the list and paired.

In type 2 (any message in the middle conversation), it can act as both a parent of the next message and a child of the previous message. Because I have been storing all possible parent messages sequentially in “conversation_texts”, I know that the parent message will be the text in position-1. I can then link the current message with the previous message and insert it into my JSON file.

Type 3 will always be the last line in a conversation so I will only need to use “conversation_texts” to find the parent and insert these linked messages into the JSON file. Here is where the conversation stops so the program will start with the next conversation. This will continue occurring until all of the conversation IDs in “movie_conversations.txt” has been read.

When running the section of code above with all the debug print messages results in the following output:

```
parent:Can we make this quick? Roxanne Korrine and Andrew Barrett are having an incredibly horrendous
public break- up on the quad. Again.child:Well, I thought we'd start with pronunciation, if that's okay
with you.
['Can we make this quick? Roxanne Korrine and Andrew Barrett are having an incredibly horrendous public
break- up on the quad. Again.', "Well, I thought we'd start with pronunciation, if that's okay with
you."]
...
```

Here I can see that I have a string parent message and a string child message. The child message looks like a suitable response. Then, the array printed underneath is the list “conversation_texts”. This shows me that the messages have been paired correctly.

The next function that needs to be made is the file insertion. In the design of my subroutines, I have proposed the idea of dumping every certain amount of linked data into the JSON file and then clearing the list of linked messages. This will allow me to do mass data dumps which may

make the creation of the training data file faster as there is a decrease in the amount of times that the file needs to be open and that data needs to be dumped inside.

Without mass dumpings, the code would look something like this:

```
def fileInsertion(parent, child): #The function takes in the string messages of the parent message and the child message after they have been paired by the pairing function
    data['intents'].append({
        'parent': parent,
        'child': child
    }) #Data is a dictionary which contains a list of intents. The intents list contains a list of dictionaries structured as below
    with open('intents.json', 'w') as outfile: #Creates the Json file (if one doesn't already exist) which we will be making our data dumps to
        #print("dumping into json") For debugs
        json.dump(data, outfile) #Dumps the data into the Json file, to be stored
```

To make a mass insertion, only a few edits to the code above must be made. For example, an if statement to check if the size of the intents list is big enough to complete the mass insertions. I have created the code below to complete this.

```
def fileInsertion(parent, child):
    data['intents'].append({
        'parent': parent,
        'child': child
    })
    if len(data['intents']) == 100: #If there are 100 intents
        with open('intents.json', 'w') as outfile:
            #print(data, "\n") For debugs
            json.dump(data, outfile)
            data['intents'] = [] #Resets the list, ready for the next mass insertion of new data
```

This code provided some problems. Although it correctly inserted the paired messages into the file, each dumping override the previous one. This meant that after each dump, the JSON file only stored 100 intents. The first conversation in the conversation file was about a girl named "Roxanne Korrin" however the first paired messages in the file were the following.

```
{
  "intents": [
    {
      "parent": "These men you up under. How you let them do ya like that?",
      "child": "Do me? Do me like this, you mean? Like chainin' me up?"
    },
    {
      "parent": "I'm gonna make you pay for that, you hear me?",
      "child": "I'm gonna make you pay for that, you hear me?"
    }
  ]
}
```

To fix this problem, I composed the following idea. Instead of completing mass dumps every x amount of intents, I will only dump the data once. When all of the data from the corpus have been linked together, I can dump it all together in the end. This prevents the file from being overridden and may decrease the time it takes to create the file as there is only one data dump.

```
def fileInsertion(parent, child):
    data['intents'].append({
        'parent': parent,
        'child': child
    })
```

The file insertion function is nearly the same as it was before except the only thing it does is create a list of dictionaries containing the linked messages. The data dumping lines have been moved to the end of the file so after the for loop (for each row in the conversation file), it will dump the data into the JSON file.

```
with open('intents.json', 'w') as outfile:
    json.dump(data, outfile)
    print("File creation complete")
```

The message printed will provide an indication on when the creation of the training data has finished. This process could range from several minutes to hours because if there are around 220,000 conversations and each conversation has at least 2 messages said, this means we are looking at several 100,000's of paired messages. With so many data needed to be stored, it could take a long time for the program to pair each one, especially because there's no efficient way for me to search for line ID's.

The following is a section from the output of the JSON message database after the training data has been created:

```
{"intents": [{"parent": "Can we make this quick? Roxanne Korrine and Andrew Barrett are having an incredibly horrendous public break- up on the quad. Again.", "child": "Well, I thought we'd start with pronunciation, if that's okay with you."}, {"parent": "Well, I thought we'd start with pronunciation, if that's okay with you.", "child": "Not the hacking and gagging and spitting part. Please."}, {"parent": "No, no, it's my fault -- we didn't have a proper introduction ---", "child": "Cameron."}, {"parent": "Cameron.", "child": "The thing is, Cameron -- I'm at the mercy of a particularly hideous breed of loser. My sister. I can't date until she does."}, {"parent": "Why?", "child": "Unsolved mystery. She used to be really popular when she started high school, then it was just like she got sick of it or something."}, {"parent": "C'esc ma tête. This is my head", "child": "Right. See? You're ready for the quiz."}, {"parent": "Right. See? You're ready for the quiz.", "child": "I don't want to know how to say that though. I want to know useful things. Like where the good stores are. How much does champagne cost? Stuff like Chat. I have never in my life had to point out my head to someone."}, {"parent": "I don't want to know how to say that though. I want to know useful things. Like where the good stores are. How much does champagne cost? Stuff like Chat. I have never in my life had to point out my head to someone.", "child": "That's because it's such a nice one."}, {"parent": "Sure have.", "child": "I
```

For easier view from the documentation, I have formatted the JSON file:

```
{
  "intents": [
    {
      "parent": "Can we make this quick? Roxanne Korrine and Andrew Barrett are having an incredibly horrendous public break-up on the quad. Again.",
      "child": "Well, I thought we'd start with pronunciation, if that's okay with you."
    },
    {
      "parent": "Well, I thought we'd start with pronunciation, if that's okay with you.",
      "child": "Not the hacking and gagging and spitting part. Please."
    },
    {
      "parent": "No, no, it's my fault -- we didn't have a proper introduction ---",
      "child": "Cameron."
    },
    {
      "parent": "Cameron.",
      "child": "The thing is, Cameron -- I'm at the mercy of a particularly hideous breed of loser. My sister. I can't date until she does."
    },
    {
      "parent": "Why?",
      "child": "Unsolved mystery. She used to be really popular when she started high school, then it was just like she got sick of it or something."
    },
  ]
}
```

The first paired messages is about Roxanne Korrine, which is the expected output. You can see above that there are different lengths in conversations (because the child message in one pair is the parent in the next one). Some conversations have 1-2 lines and others may have many more. Because there are over 200,000 conversations in the Cornell Movie-Dialog Corpus, I can assume that my JSON file contains at least 200,000 linked pairs. This means that I can teach all of these pairs to my chatbot so that it will learn different types of appropriate responses to different types of messages. The more linked pairs that are in my training data, the better the chatbot will be. At least 200,000 linked pairs in the training data is enough to have a sufficiently working chatbot.

Looking at the design for the file creation subroutine, I realised that I had forgotten to insert tags into the training data. The Tensorflow model of the chatbot will allow me to predict which category was most likely said by comparing the parents with the user's input. These categories

can't have names like "greeting" or "food" because they must be unique and with so many linked pairs, there will be a lot of conflicts between topics and a lot of computer processing will be required to place each paired link into a category. However, I don't require a context-based category name, the model doesn't care if the paired link is about food (only that each one is unique) so there's no reason for me to add code which does this. Instead, I can create incrementing tags which are unique for each paired link. The way to do this is to add a counter which increments as well as adding a variable into the dictionaries stored in intents called tags.

```
movie_lines = []
data = {}
data['intents'] = []
tag = 0 #Unique number representing a linked pair
```

The counter value "tag" can be defined as an integer, along with the other variables defined at the beginning of the Training Data module.

```
def fileInsertion(parent, child):
    global tag #By setting tag as a global variable, this function can access the
    variable and edit it
    tag += 1 #Tag increments. Since its first value is 0, this sets it to 1 so the
    first paired messages will have the tag number 1
    data['intents'].append({
        'parent': parent,
        'child': child,
        'tags': tag
    }) #New tag value in the dictionary
```

This new variable has then been used here. The "tag" variable had to be defined outside of the file insertion function otherwise the tags will always be the same when incrementing. Another solution to this would have been to create a new function which takes in an integer (the tag value) then returns an incremented value. Creating a new Training Data JSON file results in the following data being stored (after being formatted so it can be easily read):

```
{
  "intents": [
    {
      "tag": 1,
      "parent": "Can we make this quick? Roxanne Korrine and Andrew Barrett are
                having an incredibly horrendous public break- up on the quad. Again.",
      "child": "Well, I thought we'd start with pronunciation, if that's okay with
                you."
    },
    {
      "tag": 2,
      "parent": "Well, I thought we'd start with pronunciation, if that's okay with
                you.",
      "child": "Not the hacking and gagging and spitting part. Please."
    }
  ]
}
```

I have used a range of different variables in this section of code, each with different functions. To clearly show roughly what each key variable looks like and what it does, I have created the following table:

Name	Type	Size	Value
data	dictionary	1	{"intents": [{"..."}, {"..."}, {"..."}]}
movie_lines	list	304713	[['L1045', 'u0', 'm0', 'BIANCA', 'They do not!']...]
r	string	1	L1045 +++\$+++ u0 +++\$+++ m0 +++\$+++ BIANCA +++\$+++ They do not!
row	string	1	u0 +++\$+++ u2 +++\$+++ m0 +++\$+++ ['L198', 'L199']
temp_list	list	5	['L666256', 'u9034', 'm616', 'VEREKER', 'Colonel Durnford...']

Stage 1 Review

What has been done?

In order to teach my chatbot to speak using machine learning, I need to train the chatbot with some type of training data. This entire section has been the creation of my training data which will be used to train my model after it has been created. To create this training data, I used an online movie script corpus which had paired movie lines for me so that I could extract the pairs and insert them into my own database (as dictionaries in a JSON file). With this JSON file, I can easily access paired messages for the duration of the training of the chatbot model. When my chatbot is used to create a response to the user's message, it will create a prediction based on the likelihood that the user has said a certain parent in the training data. After finding the most similar parent to the user's message, the most suitable parent, the child message will be used as a response to the user.

How has this been tested?

In the earlier version of the code, I used debug print messages to print out variables such as the movie line read from "movie_lines.txt" as well as the formatting of any variable. This was done to ensure that the code is retrieving the correct piece of data from the text files to prevent any errors later on. I have also been using in-code checks to find out if the data retrieved from the text files is suitable. For example, each row in a line contains an array of data so I checked if there was the same amount of variables in each row. This would prevent errors such as using the wrong ID if the list is missing a variable.

Another check that is done in-code is for the list of conversation IDs. The code checks if there is one ID in that list and if there is, I reject this list and move onto the next row (the next list). This is done because I could get errors when trying to pair up a conversation which contains only one statement. This error would happen because the message is the first line and the last line which means that the program may interpret this line as its own parent and child.

Furthermore, I have created a variable table which shows what the most important variables look like. This allowed me to check if the code is correctly (for example) extracting data from that list or using whatever data type it is correctly. This made sure that the training data is correctly formatted so that it can be easily extracted in another section of code.

The last test was done after a few paired links were inserted into the JSON file. Just by preview, I looked over the JSON file to see: if the paired messages made any sense, if a reply to a message has the correct previous message as its response and if the JSON file is formatted

correctly (as intended). After the JSON file has passed this check, I was able to run the code to create a JSON file filled with all of the extracted paired movie lines.

How does this meet with the success criteria and/or requirements?

Training data creation is a requirement listed in the success criteria. This requirement is linked to the bot training and machine learning aspects of the success criteria as the chatbot has to be trained using some kind of training data and it won't be able to use machine learning until it has gone through some training. This criteria is met because there has been a successful creation of training data.

Changes in design due to this section

In my design, I didn't specify which type of file I will be using, instead suggesting the use of either a text file or a JSON file. This is because I was not sure of what the best way of storing data that could also be easily accessed from the file. After doing further research on these, I decided that a JSON file would suit my needs better as I could use dictionaries and arrays to store a large amount of data. This could then be easily accessed by using array indexes and dictionary keys. As well as this, the formatting of the data made extracting large amounts of data easier than in a text file. There weren't any other changes because the subroutines followed the plan.

Summary of the project so far

The training data is technically not a part of the programming part of the project, it is just data that the project will eventually be using. There is no user interface or program which opens up from the progress done in this section. This part of the project was more of a behind-the-scenes part which the user won't see or experience but it will be crucial for later parts in the project which will involve the user as discussed earlier.

Stage 2: Bot Training

In the previous stage, I have created a JSON file called Intents. This file consists of linked parent and child messages which are categorised into unique, incrementing tags. In this section, I will create and train the chatbot model using Tensorflow and TFlearn. There are several new key words that I will be using, I will begin by explaining the meaning of three. A pattern is an input message which the model will learn. These patterns are the same as the parent messages stored in the training data as these are the “patterns” that the model will learn from. A class is a category of an intent. A response refers to the response message to the user. Pattern is interchangeable with parents, classes with tags and responses with children. The language has changed a bit to suit the same language used when describing how the model works.

Before discussing what needs to be done in this section, I will explain why it was so important to layout the training data in the previous section. A chatbot framework needs a structure in which conversational intents are defined. A JSON file was used because it's a clean way to store large amounts of data that can later be extracted. Each conversational intent contains: a tag (a unique number that identifies that pair), a pattern (the parent, the patterns in the parents will be used by the neural network) and a response (the child messages will be used as responses). The pattern is what the user has most likely said and the child is an appropriate response to what was said. Therefore, by pairing up parent and child messages in the training data, I have created a way for the chatbot to learn what suitable response to a message is by comparing parents. In this way, the chatbot takes in a message to figure out what the intent in the user's message is to provide an appropriate response.

The first thing that needs to be done is reading the training data from “intents.json”. “intents.json” is a very large file because there is a large amount of data that is stored inside. The last tag in the file is number 138519, that means that the training data has 138,519 linked messages. This implies that although the corpus used had 200,000+ movie conversations, not all of them were actually used in the “movie_conversations.txt” file leading to less pairs being made.

The following are the imports needed for this section:

```
#All NLTK imports are for natural language processing
from nltk.tokenize import word_tokenize #This allows for tokenizing words in a sentence
- splitting of words in a sentence into a list of words
import json
import string #Used to provide their built-in list of punctuation
import numpy as np #For the use and conversion of numerical data
import random #To randomise the patterns (the parents)
import tflearn #A deep learning library that sits on top of Tensorflow (it's a higher
level API)
import tensorflow as tf #Library for machine learning
import pickle #For saving the data structures created in this section
```

After the imports, the first thing that needs to be done is loading the training data (the intents JSON file) into the program. I will be using the “json” module to read through the “intents.json” file so that I can later loop through the data and use it. I will be using this to train the model.

```
with open('intents.json') as json_data:
    intents = json.load(json_data) #Loads the data inside the Json file as a list of
dictionaries
```

Note that the intents variable is a dictionary itself which contains a list of dictionaries (the intents themselves). This means that this variable stores all of the contents of the file inside. There are three important lists that need to be defined, words, classes and documents. After the imports and these two lines of code, the file has been read. The next thing that needs to be done is preprocessing the data. Currently, the data cannot be taught to a model because the model uses numerical data however the training data is composed of strings. This means that I have to do some data processing to remove certain parts from the strings that aren’t needed (for example punctuation) and then convert it into the correct type of data that a Tensorflow model would be able to learn from.

```
punc_dic = dict((ord(punct),None) for punct in string.punctuation) #creates a
dictionary which I can use to remove punctuation
words = [] #List of all words used
classes = [] #List of all the classes
documents = [] #Corpus containing parents with their tags
```

The first variable defined here is “punc_dic”. This is a dictionary that contains all of the punctuation stored by the “string” library. For each punctuation in the “string” library, the punctuation is turned into unicode by the “ord()” function and stored with the value “None”. This dictionary looks like the following.

```
{33: None, 34: None, 35: None, 36: None, 37: None, 38: None, 39: None, 40: None, 41: None, 42: None, 43: None, 44: None, 45: None, 46: None, 47: None, 58: None, 59: None, 60: None, 61: None, 62: None, 63: None, 64: None, 91: None, 92: None, 93: None, 94: None, 95: None, 96: None, 123: None, 124: None, 125: None, 126: None}
```

This will later be used to remove punctuation from the patterns. The words list will be a list that stores all words used in the patterns. It should not have any duplicate words so this list will need to be sorted after all words are inserted. The classes list will store all of the tags in the training data. The documents list will act as a corpus, it will store each pattern along with its class.

Now I want to add the data needed into the three lists created earlier.

```
for intent in intents['intents']: #Loops through all of the intent dictionaries in the training data
    pattern = intent['parent'] #Pattern equals the parent message in the current dictionary in the for loop
    w = word_tokenize(pattern) #Pattern is tokenized so w is a list of words
    words.extend(w) #Adds to the word list
    documents.append((w, intent['tag'])) #Adds documents to our corpus
    if intent['tag'] not in classes: #If the class isn't already in the list of classes
        classes.append(intent['tag']) #Adds to the classes list
```

Using a for loop, I iterate through each intent (each dictionary) in the training data. I set the variable pattern to be the parent message. This is then tokenized. This tokenization is part of Natural Language Processing and it's a feature of NLTK. What tokenization does is it splits up a sentence into a list of 'tokens'. For example, if I said "Hello, my name is Danielle!" and tokenized this, the resulting value would be ["Hello", ",", "my", "name", "is", "Danielle", "!"]. "words.extend(w)" adds the new list of tokens into the list of words. This is the same as using a for loop on the tokenized list (w) and appending each value into the list. This process is done because I want to know all of the words used in the patterns in the intents file, which will be important later on. The list of tokenized words and the class/tag is appended into the documents list. If the class is not in the list of classes (which it shouldn't be the case as each intent has a unique, incrementing number which identifies it but this is done just in case there are errors in the training data file) then the class will be added into the list.

After the section of code above is complete, the "words" list will contain a large amount of words, with a lot of duplicates. However, I want a list of all the words used so having duplicate words ruins this list so as I mentioned earlier, the duplicate words must be removed. Another problem is punctuation. There are two occasions where I must remove punctuation from messages: 1. Remove punctuation from the patterns before training the chatbot 2. From the user's message before comparing it to the patterns. This is important because if the chatbot is

trained on messages with punctuation, it will learn to differentiate two of the same message as different due to the lack of punctuation in one. For example, if I said “Hello!” to the chatbot, it should get the same prediction and response as if I said “Hello”. The only difference between these two is the exclamation mark which makes no difference on the meaning.

```
words = [word.lower().translate(punc_dic) for word in words] #Turns all of the words  
into lower case and removes all punctuation by replacing it with a blank string ''
```

Removal of punctuation is done in one line in the code above. For each word in the list “words”, the word is converted to lowercase and all punctuation is translated using the punctuation dictionary created earlier. What this does is replace any punctuation used in the word with a blank string ”. This new list created then replaces the previous values in the list so the list is now fully lowercase with no punctuation.

```
words = sorted(list(set(words))) #Removes duplicated words in the list then sets it  
back to a list that's sorted which means that the blank characters replacing  
punctuation only appear once (at the first index)
```

In the line above, duplicated words are removed from the list. First, “set()” removes any duplicated words so there is only one word per word used in patterns. This then needs to be set back into a list (as the new value is no longer a list) because set is its own data type. This new list will then be sorted into alphabetical order. The blank character which used to represent punctuation is now present at the first index of the “words” list, index 0. A blank character is not a word that needs to be taught to the model so this value is removed by the line of code below:

```
words.remove('') #I can simply remove the character that used to be punctuation
```

Now, the “words” list is a large list of words with each value being a unique and non-repeating word with no punctuation, all in lowercase. Ideally, this would be suitable data to change into numerical data. However that is not the case. To show the problem, I will give an example: I, a user, say “I run home!” to the chatbot. With the current code, the exclamation mark is removed and the “I” is turned into lowercase “i”. Assuming that the model is all complete and the predictions work, the model will predict a response for me. However, if I said “I ran home.”, the full stop will be removed and the “I” will also be turned into a lowercase “i” and the model will provide me with a prediction. These two statements have the same meaning, it’s only different tenses that are being used. However, the model would predict different suitable responses for these messages because “run” is not the same as “ran”. This scenario can happen many times, when a word is used in different ways according to tenses or how it was used grammatically which can cause inaccurate predictions. To prevent this, I can use natural language processing

from the library “NLTK” to convert the words into their root words, before training and before making predictions.

There are two ways of turning a word back into its original root, stemming and lemmatization.

Stemming, by definition, is the process of reducing reflection words to their root forms. There are two different English stemmers that NLTK provides, PorterStammer and LancasterStammer. PorterStammer is the older stemmer between the two. These two stemmers work differently. The PorterStammer uses Suffix Stripping (the process of removing suffixes from a word to get the root word). When using both of these stemmers to stem two words “cats” and “trouble”, the results are “cat” and “troubl”. The word “cats” has successfully been stemmed into the correct root word, but this isn’t the case with the 2nd word. “Troubl” is not a English word so the stem is not correct. The main benefits of using the PorterStammer is that it’s known to be simpler and faster than the LancasterStammer. The LancasterStammer is an iterative algorithm which has external rules. On each iteration, it tries to find an applicable rule by the last character of the word. This usually results in further stemming of the word. On the other hand, the LancasterStammer tends to over-stem, which will also result in a stem word which has no meaning (just like with the case of using the word “trouble” before).

Lemmatization is defined to be the process of stemming the word in a way that also takes into account that the resulting root (which is called a lemma in lemmatization) word is an actual word in the language. Python NLTK provides the WordNet Lemmatizer (this was seen imported earlier in the code). Using the same examples used in stemming, if I wanted to create lemmas from the words “cats” and “trouble”, the resulting words would be “cat” and “trouble”. However, the biggest draw back of lemmatization is that you need to specify whether the word is a noun, adjective, verb or adverb. This means that I will need to use some kind of algorithm to determine whether a word is one of these four.

There are big drawbacks with both processes. I want to conduct a test before choosing which process I will use because it doesn’t matter if the chatbot learns from real root words or not (because if I use the same stemmer/lemmatizer the resulting root word will always be the same). What does matter is how well it turns a large amount of words into as least unique words as possible to decrease the amount of the same words (like walk, walking and walked) that are taught to the chatbot. My solution to solve the draw back of lemmatization is by using NLTK’s part-of-speech tagging. Part-of-speech tagging is the process of tagging each word in a sentence with its grammatical value (whether it’s a noun, verb, adjective or adverb) according to the context in the sentence. This means that after using position tagging on the words, I can compare the tags with ones known to represent e.g. nouns and then lemmatize the word with the setting of a noun. Below is the code which does this.

```

def lemmatizer(words): #This will work for both multiple words and 1 word
    lem = WordNetLemmatizer() #Sets the lemmatizer to be NLTK's WordNetLemmatizer
    lemmed = [] #A list of lemmas
    tagged = pos_tag(words) #A tuple containing sets of words with their part-of-speech
    tags per word
    for pair in tagged: #For each pair in the tagged list (each set of word with its
    position tag)
        if pair[1] == "NN" or pair[1] == "NNS" or pair[1] == "NNP" or pair[1] == "NNPS"
        or pair[1] == "PRP" or pair[1] == "PRP$" or pair[1] == "WP": #If the word is a
        noun or a word type similar to a noun
            root = lem.lemmatize(pair[0], pos = 'n') #Lemmatize the word with the noun
            setting
        elif pair[1] == "DT" or pair[1] == "JJ" or pair[1] == "JJR" or pair[1] == "JJS"
        or pair[1] == "PDT" or pair[1] == "WDT": #If the word is an adjective or a word
        type similar to an adjective
            root = lem.lemmatize(pair[0], pos = 'a') #Lemmatize the word with the
            adjective setting
        elif pair[1] == "VB" or pair[1] == "VBD" or pair[1] == "VBG" or pair[1] ==
        "VBN" or pair[1] == "VBP": #If the word is a verb
            root = lem.lemmatize(pair[0], pos = 'v') #Lemmatize the word with the verb
            setting
        elif pair[1] == "RB" or pair[1] == "RBR" or pair[1] == "RBS" or pair[1] ==
        "WRB" or pair[1] == "MD": #If the word is an adverb or a word type similar to
        an adverb
            root = lem.lemmatize(pair[0], pos = 'r') #Lemmatize the word with the
            adverb setting
        else: #If it has a different tag or was not tagged at all
            root = pair[0] #The word will stay the same
        lemmed.append(root) #Appending the lemmas to the list of lemmas
    return lemmed #Return the list of lemmas

```

The code above codes for a lemmatization function. The lemmatizer takes in a list of words (the tokens after a sentence has been tokenized). Then these words go through part-of-speech tagging, forming a tuple which stores the word and the tag together in a pair. Next, I use a for loop to iterate through all of the pairs and run checks on the word, if it was tagged as a noun, adjective, adverb or verb then it should be lemmatized in that way. Else, if it was not tagged at all or its tag does not fit any of the criteria then the lemma is the word. I create a list of all the lemmas created in this process which is then returned.

To make this function work, the following libraries must be imported:

```
from nltk import pos_tag #Part-of-speech tagging allows for contextualisation. Words in  
a sentence can be appropriately and grammatically tagged based on context (for example,  
a word that's a verb and a noun, you would use the context it's used in to determine  
whether the word is a verb or a noun)  
from nltk.stem import WordNetLemmatizer #Provides a lemmatizer for lemmatization
```

This function can be used as shown below:

```
words = [word.lower().translate(punc_dic) for word in words]  
words = sorted(list(set(words)))  
words.remove('')  
words = lemmatizer(words) #Words are lemmatized
```

There are two problems encountered after doing this. Firstly, the list has been sorted from any duplicate words and only after lemmatized. However, if there are different words e.g. “run”, “ran”, “running”, these would stay as different words when sorted however when lemmatized, it would produce duplicates of the word “run”. This means that the words must be lemmatized before they are sorted. Another problem is that words is a big list of words with thousands of sentences that have been combined together. This means that using POS tagging is a bit useless because some words will not be in the correct context. To solve both these problems simultaneously, I need to lemmatize the patterns before the words are added into the “words” list so that they are correctly lemmatized and can then be sorted with no duplicate words.

```
for intent in intents['intents']:  
    pattern = intent['parent']  
    w = word_tokenize(pattern)  
    w = lemmatizer(w) #Lemmatize the words in context, before they can be sorted  
    words.extend(w)  
    documents.append((w, intent['tag']))  
    if intent['tag'] not in classes:  
        classes.append(intent['tag'])  
  
words = [word.lower().translate(punc_dic) for word in words]  
words = sorted(list(set(words)))  
words.remove('')
```

This was done here. The patterns are lemmatized before they are added to the “words” list so that when this list is sorted, it will sort all duplicate words (including those caused by the lemmatization). To test if this has worked correctly, I will be using the following debug statements to show if the code has worked correctly.

```
#DEBUG TESTING
print(len(documents), "documents")
print(len(classes), "tags", classes)
print(len(words), "unique stemmed words", words)
```

I am expecting 138,519 classes and documents. There is no expected number of words because I have no way of estimating how many words each pattern has. When running this code, the following is the output:

```
138519 documents
138519 tags [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30,
31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61,
62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92,
93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118,
119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143,
144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168,
169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193,
'unsuccessful', 'unsuitable', 'unsung', 'unsupervised', 'unsupported', 'unsure', 'unsuspecting', 'unt', 'untermeyer',
'unthinkable', 'untie', 'until', 'untimely', 'unto', 'untouchable', 'untouched', 'untraceable', 'untrained', 'unteaceable',
'untreated', 'untrue', 'untrustworthy', 'untruth', 'unturned', 'ununderstanding', 'unusual', 'unusaldoes', 'unusually',
```

The expected outcomes for tags and documents is correct. In the screenshot, I have shown the beginning of the tag section to show that it worked because the output was too large to show. I have also shown a section of all the words which show that show lemmatization has occurred. There were a total of 31,667 unique lemmatized words. For words that weren't fully lemmatized, the same process would be done to the user's input, so the same word will be used as an output. The lemmatization process was effective in that it reduced the amount of duplicated words (words that have the same meaning) that the model would have to learn. This finished "words" list contains a list of all the words that the chatbot model has learnt.

For testing sake, I will use stemming to compare between the amounts of stemmed words and lemmatized words. I want to use the least amount of words as possible because teaching the model more words unnecessary words will create unnecessary work on the neural network. I will be using the LancasterStemmer and the following code to test the difference between stemming and lemmatization on the amount of unique words.

The output of stemming was 138,519 documents, 138,519 tags and 24,969 unique stemmed words. This is 24,969 words versus 31,667. This implies that with lemmatization, the chatbot will learn 6,698 extra words which have the same meanings which indicates that using stemming would be faster. This is because less words need to be turned into bag of words (which I will explain in the next paragraph) so there will be less bag of words that the model will have to learn and use. With a difference of thousands, this could have an effect on the performance of the model. All of the code done for this section so far plus the new code for stemming can be seen below.

```
from nltk.tokenize import word_tokenize
from nltk.stem.lancaster import LancasterStemmer
import json
import string

punc_dic = dict((ord(punct),None) for punct in string.punctuation)
words = []
classes = []
documents = []

with open('intents.json') as json_data:
    intents = json.load(json_data)

for intent in intents['intents']:
    pattern = intent['parent']
    w = word_tokenize(pattern)
    words.extend(w)
    documents.append((w, intent['tag']))
    if intent['tag'] not in classes:
        classes.append(intent['tag'])

stemmer = LancasterStemmer() #Allows usage of the LancasterStemmer
words = [stemmer.stem(word.lower()) for word in words] #Stems the lowercase version of
the words using the LancasterStemmer
words = [word.translate(punc_dic) for word in words] #Same as before without the .lower()
() as this is done in the line above
words = sorted(list(set(words)))
words.remove('')

#print(len(documents), "documents")
#print(len(classes), "tags", classes)
#print(len(words), "unique stemmed words", words)
```

Note that the line of code “from nltk.stem.lancaster import LancasterStemmer” is the code which imports the Lancaster Stemmer into this code, allowing it to be used.

As mentioned earlier, this “words” list can’t be used by the model yet because the list contains string data but a neural network takes in numbers. To convert the string into suitable data, I will construct a bag of words. Each sentence is going to be represented with a list the length of the amount of words in the list of total words. Each position in the list will represent a word in the “words” list. I am going to be using 1’s and 0’s to mark these positions. A position in the list with a 1 means that the word exists in the sentence. If it is marked with a 0, then the word is not present. This is a bag of words because the order in which the words appear in the sentence is lost. The only thing known is the presence of the words from the words list (which represents

the model's vocabulary). Because each bag of words will have the same length as the "words" list, the size of each bag of words has greatly decreased due to the use of stemming instead of lemmatization. The next section of code will do this.

```

training = [] #A 2D list which will store the training data
output_empty = [0] * len(classes) #Empty array for the output_row

for doc in documents: #For each pair of pattern with its class in the corpus
    bag = [] #This will be the bag of words
    parent_words = doc[0] #List of tokenized words from the parent
    parent_words = [stemmer.stem(word.lower()) for word in parent_words] #Stemming the
    words in the same way as in the chatbot's list of words (the words list)
    parent_words = [words.translate(punc_dic) for word in parent_words] #Removing
    punctuation in the same way because punctuation is not in the words list
    for w in words: #For each word in the words list
        bag.append(1) if w in parent_words else bag.append(0) #Creates the bag of
        words array by inserting 1 where the word is present

    output_row = list(output_empty) #Output_row is equal to the empty array
    output_row[classes.index(doc[1])] = 1 #Places a 1 in the array where the class is
    present according to its index in the documents corpus

    training.append([bag, output_row]) #Appends the bag of words and the bag for
    classes into the training list

random.shuffle(training) #Randomises the order of the lists inside training so it's no
longer in order (it has been shuffled)
training = np.array(training) #Converts the list to a numpy array

train_x = list(training[:,0]) #Splits list at index 0, which is the bag of words
train_y = list(training[:,1]) #splits list at index 1, which is the bag for classes

print("x:", train_x) #For debug testing
print("y:", train_y)

```

Firstly, I have defined new variables that will be used, "training" and "output_empty". "Training" will be a 2D list which stores the bag of words in the first index of a list and a bag of classes at the second index. "Output_empty" is an empty list which will be used to contain the position of the class of the pattern which has been converted to the bag of words. I iterate the documents list so that I can get the pattern and class for every intent. The words used in the patterns need to undergo the same conversion as the words in the "words" list have to match. Once they have gone through that, each word in the pattern is checked against the words in the "words" list. If the word from the word list is found, the bag will add a 1 at that position, marking that this word is in the pattern. Otherwise, there will be a 0. Using the "output_empty" list created at the

beginning of this section, I create another similar bag but for classes. The index of the tag number (the class) stored in the corpus is found. The 0 at this index in the “output_row” list is found and converted to a 1. The “output_row” and the bag of words are appended into the “training” list in their own list, with each list marking a different pattern. This list is then shuffled so all of the numerical data for each pattern are at different orders. The “training” list is converted into a numpy array so that it can be used by TFlearn (it was previously just a list but numpy converts it into an array which is suitable for use by TFlearn). The training list is split into an x value and a y value, with the x value being the bag of words and the y value being the bag for classes. To test if these were done correctly, I have printed out “train_x” and “train_y”. Each value should have only 0’s and 1’s and should be the correct length of all the classes and words used according to the list.

Above is a section of the output. This particular output was for train_x, as that was the first value printed out. Each bag contained a lot of numbers so I couldn't count each bag to make sure that it's around 130,000 long. However, I could see that the first bag consisted of only 0's and 1's so I can assume that the bags were created correctly. Since a pattern will only have a certain number of words out of the thousands in the "words" list, there would be a lot more 0's in each bag than 1's.

The final part of this section is creating the model itself. The “train_x” and “train_y” variables are the numerical data that will be used to teach the model. In the next section, when creating user responses using this model, the user’s input message would have to be converted into a similar bag of words so that the trained model can determine context and form a response. More specifically, the way this model will work is it would take in a bag of words (made using the

user's input or the training data) and output a bag of classes and a prediction. Thus, this model will predict the likelihood that the pattern in that class was said. To create a model, I will use Tensorflow and TFlearn. The code which creates this model is below:

```
tf.reset_default_graph() #Reset the underlying tensorflow graph data
net = tflearn.input_data(shape=[None, len(train_x[0])]) #Building the neural network to
take in the bag of words created above each time
net = tflearn.fully_connected(net, 8) #This line will create a fully connected layer
with 8 hidden units
net = tflearn.fully_connected(net, 8)
net = tflearn.fully_connected(net, len(train_y[0]), activation='softmax') #Adds the
output layer
net = tflearn.regression(net) #The final layer of the network. This is an estimator
layer.

model = tflearn.DNN(net, tensorboard_dir='tflearn_logs') #Defining the model and
setting up the tensorboard
model.fit(train_x, train_y, n_epoch=1000, batch_size=8, show_metric=True) #Starts the
training and applies the gradient descent algorithm
model.save('model.tflearn') #Saves the model
```

The line “tf.reset_default_graph()” completely resets the underlying Tensorflow graph, so that it's empty. This removes nodes that have been previously added which could affect the chatbot model. Next, the neural network is set to take in the “train_x” that was created earlier in this section of code. To do this, I have created an input layer for the neural network where the shape is set to expect to be taking in data in the length of the bag of words (which will always be the same length as this is the length of the words list). The next thing I do is create two layers of eight hidden units. I have created two layers of hidden units because the hidden layers transform the inputs into something that the output layer can use. The number of hidden units is chosen according to the size of the input layer and the size of the output layer, it has to be between them. The first hidden layer of the neural network starts up the input data layer. The next layer to be created is the output layer. This was optimised for classes because those are the output of the model. Setting the “activation” to “softmax” is one of these customisation. Having “softmax” as the activation setting allows the model to get probabilities for each output. This will be used for predictions in the next section. The “len(train-y[0])” part means that the neural network will have units for each class. The final layer of the neural network is the estimator layer. In my case, I will apply a regression on the input. The regression requires the use of a TensorFlow gradient descent optimizer that will minimize the provided loss function (which calculates the errors). The required input for regression is only a 2D tensor layer (“net”). I have not defined an optimiser to use so Tensorflow will use the default one, adam. Now that the neural network has been created and set, it can be assigned to a new model. DNN stands for Deep Neural Network. The “tensorboard_dir” sets the directory to store tensorboard logs.

Now that the model has been set up, it needs to be trained. This is done by “model.fit”, which “fits” or passes the training data to the model. The number of epochs set is the amount of times (1000 times in this case) that the model will see the same information while training. Hopefully seeing the same data 1000 times will make it classify the data better. To potentially improve the model, if needed, this epoch number can be increased and played around with. “Show_metric” allows for debugging as the model is being trained so I can see in console what’s going on. Finally, this model is saved into the file “model.tflearn”.

One of the outputs from “show_metric” being enabled is the ability to see the accuracy of the model. I want to know what this accuracy is, if it’s above 80-90%, the model should be accurate enough to be used. However, because of the huge amount of data being used, the accuracy may greatly increase thus I need to run this code and test out how accurate the model created is (by its calculated scale).

This entire code will now run together to create a neural network. I am expecting there to be a huge amount of outputs due to the epochs so I will not be able to provide what the entire output looks like. However, I will be able to show segments of what’s happening.

After running the code, I have received an error - “AttributeError: ‘list’ object has no attribute ‘translate’” on “parent_words = [words.translate(punc_dic) for word in parent_words]”. “Words” is the word list, so translating that list would indeed cause the AttributeError. This was due to a typo, “words.translate(punc_dic)” needs to be replaced with “word.translate(punc_dic)” because the “parent_words” list is being iterated through with the variable “word” being the current word taken from the list.

With the new change, the code starts up running smoothly but when it gets to the line “output_row = list(output_empty)”, the program halts due to a MemoryError exception. This error is due to the computer running out of memory in the RAM for the code to execute, which makes sense to have occurred considering it’s handling ~24,000 words and there are ~24,000 bag of words with the length of ~24,000, as there is a bag for each word. As well as this, the class list is ~138,000 long which means that along with the very long bag of words, there are ~24,000 class bags where each one is ~138,000 long. From this, the error can be seen, there is just too much data for the majority portion of 16GB of RAM dedicated to Python to handle. The obvious option of getting rid of this error is by decreasing the amount of data in the training data. However, I don’t want to decrease the training data because I want the chatbot to be as efficient as possible but with less data being used, it will affect performance, so I want to try to find alternative solutions. To look at ways to fix this, I firstly looked at my “words” list:

```
'zzzzzzzzzzzzzzzz', '\x8ccause', '\x8cem', '\x8chood', '\x91art\x92\x94', '\x91cause', '\x91closed', '\x91em', '\x91haveit',
'\x91my', '\x91no\x92', '\x91round', '\x91sleeping', '\x91that', '\x92', '\x93hi\x94', '\x93i', '\x93la', '\x93ladies',
'\x93leon', '\x93never', '\x93not', '\x93well', '\x94', '\x96', '\x96nice', '\x96\x96', '\x97', '\x97a', '\x97and',
'\x97away', '\x97before', '\x97buy', '\x97closer', '\x97extanker', '\x97for', '\x97glad', '\x97goes', '\x97i', '\x97in',
'\x97it', '\x97mr', '\x97running', '\x97that', '\x97the', '\x97then', '\x97there', '\x97this', '\x97whoever', '\x97with',
'\x97word', '\x97you', '\x97\x97', '\x97\x97broke', '\x97\x97doors', '\x97\x97frightening', '\x97\x97nobody', '\x97\x97the',
'\x97\x97walk', '\x97\x97you', '\x97\x97young', '£1000', '£150', '£15000', '£20000', '£3106', '£4000', '£our', '«', 'à',
'quelqu\x92un', 'ça', 'éisé', 'ête', 'être']
```

```
, your cyoo, yours, your scii, your scii wyyttoo6, your scii, your scii
'you\x92d', 'you\x92ll', 'you\x92re', 'you\x92ve', 'you\x97', 'you\x97you'
', 'yoyo', 'yoyoyiggityyo', 'yready', 'ysee', 'ythink', 'ytryin', 'yu', '
```

```
24969 unique stemmed words [ '0', '007', '009843', '0200', '0300', '05', '0504', '06', '0630', '07410', '0800', '0821',
'0900', '0h', '1', '10', '100', '1000', '10000', '100000', '1002', '100page', '101', '1012', '101st', '1030',
'104', '105', '105353', '1060', '108', '109', '1098', '10b', '10th', '10yearold', '11', '110', '1100', '110th', '1115',
'1130', '1138', '115', '119', '11th', '12', '1200', '12000', '120000', '120130', '1208', '1209', '1215', '122', '124', '125',
'127', '1274', '1292', '129467b', '12gauge', '12th', '12year', '13', '130', '132', '1320', '1322', '133', '134', '135',
'137', '138', '13dollar', '13th', '14', '140', '14000', '1412', '1425', '143', '1436', '14380', '1443', '146', '147', '1490',
'14b', '14th', '14yearold', '15', '150', '1500', '15000', '1530', '155', '156', '15th', '16', '160', '161', '167', '1672',
'16m', '16mm', '16th', '17', '1700', '1713', '1787', '179', '1791', '17c', '17hundred', '17th', '17year', '18', '180',
'1800', '18000', '180000', '1800deepsea', '180th', '18250', '183', '1835', '184', '184000', '1852', '1854', '1856', '1858',
'1862', '1867', '1868', '1870', '1871', '1872', '1879', '188', '1888', '1890', '1893', '1894', '1895', '1896', '1898',
'1899', '18hundred', '18k', '18th', '19', '190', '1904', '1905', '1914', '1915', '1916', '1917', '1919', '1920',
'1920s', '1921', '1922', '1927', '1930', '1932', '1935', '1936', '1937', '1938', '1939', '1940', '1941', '1942', '1945', '1946',
'1947', '1948', '1949', '195', '1950', '1952', '1953', '1956', '1957', '1958', '1959', '1960', '1962', '1963', '1964',
'1965', '1966', '1967', '1968', '1969', '1970', '1971', '1972', '1973', '1974', '1975', '1977', '1978', '1979', '198',
'1980', '1981', '1983', '1984', '1985', '1986', '1988', '1989', '199', '1992', '1993', '1995', '1996', '1997', '1998',
'19th', 'legal', 'love', '1st', '2', '20', '200', '2000', '20000', '200000', '2001', '2004', '2005', '201', '2016', '2017',
'202', '2020', '2034', '205', '20s', '20th', '21', '2100', '210pound', '211', '212', '218', '21st', '22', '2224', '22350',
'2245', '225', '22nd', '23', '230', '2300', '235', '237', '23763273', '23rd', '24', '2400', '241', '2415', '247', '24hour',
'24k', '24thcentury', '25', '250', '2500', '25000', '250000', '253', '254', '255', '256', '25page', '25s', '25th',
'25yearold', '26', '260', '2627', '2684', '26th', '27', '270', '2702', '2735', '275', '27b6', '28', '280', '289', '28th',
'29', '290', '2983', '299', '29th', '2nd', '2page', '2x', '2x4', '3', '30', '300', '3000', '30000', '300000', '300w',
'300th', '302', '303', '304', '3054410', '306', '3089962', '30th', '31', '310', '31259', '314', '315', '316', '318', '31st',
'32', '32000', '325', '325th', '327', '33', '330', '3350', '337', '337099876', '33rd', '34', '3407', '3417', '342834',
'3444', '347', '34th', '35', '350', '3500', '35000', '350000', '350400', '357', '36', '364000', '365', '37', '37000', '372',
'375', '38', '383', '3860', '38ths', '395', '3jane', '3rd', '4', '40', '400', '4000', '40000', '400foot', '403',
'4044532222', '4060', '409', '41', '410', '411', '4121', '414', '415', '41st', '42', '4215', '422', '425', '426', '42nd',
```

These are three scenarios that I found where space in the “words” list is wasted. In the first picture, the words said aren’t even English but the chatbot is, also currency doesn’t need to be analysed by the chatbot as it’s highly unlikely that it will affect the prediction greatly. In the second image, there are more non-English words being used. In the third image, there are so many numbers being stored in the “words” list. Overall, it looks like numbers and words with non-alphabetical characters take up a lot of space in the “words” which increases the size for each bag of words. This means that adding a check before creating the bag of words may prevent the memory error. A simple check could be iterating through the “words” list and checking if the word contains only symbols from the alphabet.

```
for i in words: #Iterates through the word list, i is a word
    if i.isalpha() == False: #Uses Python's method to check if the string contains all
        alphabetic numbers
    words.remove(i) #Removes the word
```

Here, I use Python's built-in function to check if the string contains only alphabetic characters. The output is now 24,242 uniquely stemmed words. However the original count is 24,969. The roughly 700 words lost is insignificant, especially because the words list still contains a lot of wasted space. As well as this, I found that some words that are fully alphabetical were removed, which makes the “words” list lose unique keywords for example with the words “steady”, “started”, “tell”, “taken” and “the”.

I tried different ways of removing useless words from the “words” list:

```
unwanted = ["'", "0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "£", "$"] #A list of unwanted characters that strings may contain. If a word has any of these, it should be removed from the words list
for i in words:
    for x in unwanted: #For each character in the unwanted list
        if x in i: #If the character is in the word
            words.remove(i)
            break #Stop the check there because the word has already removed so it doesn't matter if it contains more unwanted characters
```

This method resulted in 24,571 uniquely stemmed words.

```
for i in words:
    for x in unwanted:
        if i.find(x) != -1: #If there's an index for the character because it has been found in i
            words.remove(i)
            break
```

This method resulted in 24,556 uniquely stemmed words.

When checking the “words” list during the debugging of this method, I found that words such as “£1500”, “00”, “1917” and “hello\x97” were present in the list. This means that these methods have not effectively removed the words wanted. It has also occurred to me that removing words from the words list could cause errors in the bag of words as some words will not be added to the bag, so this will not be a viable solution. Another possible solution is splitting up this code into two or three modules (because it's very likely that this memory error will occur during the actual training as well). This is because the memory runs out for the session but if there are multiple sessions running at different times, this would prevent the error (because the program has been split up). However, I will have to make sure that each module can access the data from each other so that it can work as one. This can be done using the “pickle” library. Firstly, I am going to split up this code like so in one separate module:

```
from nltk.tokenize import word_tokenize
from nltk.stem.lancaster import LancasterStemmer
import json
import string
import pickle

punc_dic = dict((ord(punct),None) for punct in string.punctuation)
words = []
classes = []
documents = []

with open('intents.json') as json_data:
    intents = json.load(json_data)

for intent in intents['intents']:
    pattern = intent['parent']
    w = word_tokenize(pattern)
    words.extend(w)
    documents.append((w, intent['tag']))
    if intent['tag'] not in classes:
        classes.append(intent['tag'])

stemmer = LancasterStemmer()
words = [stemmer.stem(word.lower()) for word in words]
words = [word.translate(punc_dic) for word in words]
words = sorted(list(set(words)))
words.remove('')

pickle.dump( {'words':words, 'classes':classes, 'documents':documents}, open(
    "memoryerrordump", "wb" ) ) #Dumps the important variables into a pickle file called
    "memoryerrordump" so they can be retrieved
```

This is the first module. Here, all of the data extraction and formatting of the chatbot's word list occurs. The important variables are pickled so they can be used in this next module:

```

from nltk.stem.lancaster import LancasterStemmer
import string
import pickle

training_data = pickle.load(open("memoryerrordump", "rb")) #Open the variables created
in the previous section
words = training_data['words']
classes = training_data['classes']
documents = training_data['documents']
punc_dic = dict((ord(punct),None) for punct in string.punctuation)
stemmer = LancasterStemmer()
training = []
output_empty = [0] * len(classes)
doc_counter = 0 #Counts the document number it's currently on during the iteration

for doc in documents:
    if doc_counter != 18001: #If it has not done this process on 18000 documents yet (the
highest count I found that doesn't bring any MemoryError exceptions)
        doc_counter += 1
        bag = []
        parent_words = doc[0]
        parent_words = [stemmer.stem(word.lower()) for word in parent_words]
        parent_words = [word.translate(punc_dic) for word in parent_words]
        for w in words:
            | bag.append(1) if w in parent_words else bag.append(0)

        output_row = list(output_empty)
        output_row[classes.index(doc[1])] = 1

        training.append([bag, output_row])
    elif doc_counter == 18001: #If it has done this process on 18000 documents
        break #Break out of the For loop
pickle.dump( {'words':words, 'classes':classes, 'documents':documents,
'training':training, 'counter':doc_counter}, open( "trainingsdump1", "wb" ) ) #Pickles
all of the important variables so they can be retrieved

```

This code will occur in a separate module to the first. Here, the variables created in the first are used to create a bag of words for the first 18,000 documents in the “documents” list. The numerical data created for these documents are then pickled so they can be saved. I will have to create separate pickles for each set of 18,000 words until they have all been done. This code will only need to run for the first 18,000, afterwards, it has to be modified so it can do this for the rest of the data. Of course, this is not an ideal way to create a chatbot but because I am trying to teach so many intents to the chatbot, it exceeds the 16GB of RAM on my computer so this is the best way that I found to still use my intents. The module below is an adapted version of the

above (because the above only needs to occur once). This will need to run multiple times until all the documents have been turned into numerical data.

```

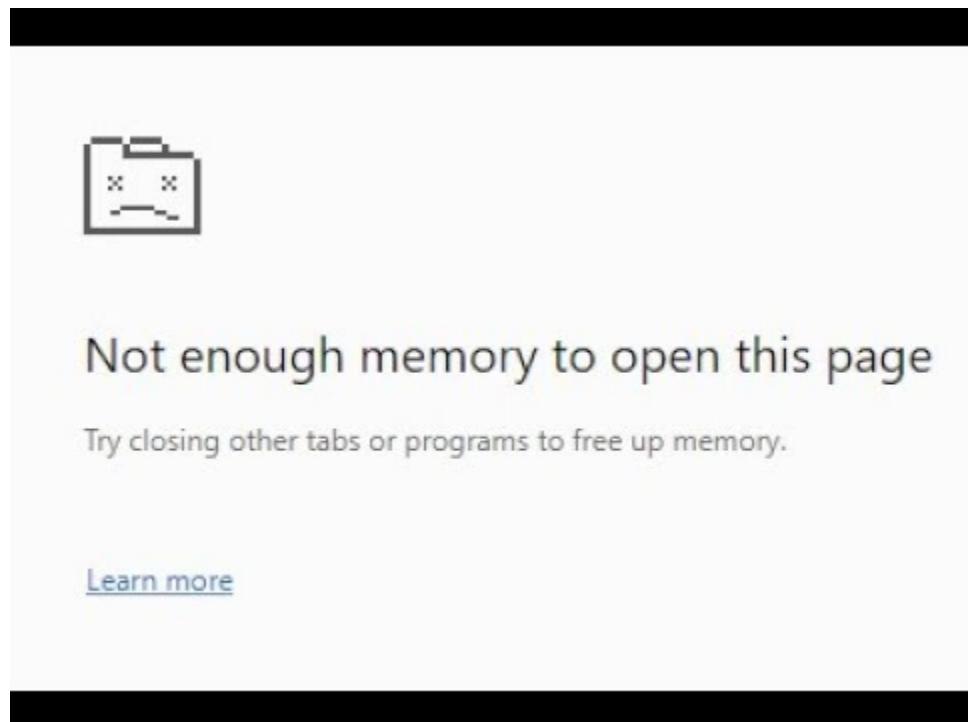
from nltk.stem.lancaster import LancasterStemmer
import string
import pickle

iteration = 2 #For counting the new file name to be made and the previous file to read from
open_name = "trainingdump" + str(iteration-1) #The name of the file to read from
training_data = pickle.load(open(open_name, "rb"))
words = training_data['words']
classes = training_data['classes']
documents = training_data['documents']
doc_counter = training_data['counter'] #Counter from the previous document so the bag of words are created for the next set of documents after this number
punc_dic = dict((ord(punct),None)) for punct in string.punctuation
stemmer = LancasterStemmer()
training = []
output_empty = [0] * len(classes)
max_count = doc_counter + 15001 #There was memory errors with running + 18001 so attempting 15001
pos_counter = 0 #Counts the position of the documents so that I can determine at which point the program has reached the documents to process

for doc in documents:
    if pos_counter == doc_counter: #If the position is the processed documents from the last iteration
        if doc_counter != max_count: #If the next set of documents haven't been processed yet
            doc_counter += 1
            bag = []
            parent_words = doc[0]
            parent_words = [stemmer.stem(word.lower()) for word in parent_words]
            parent_words = [word.translate(punc_dic) for word in parent_words]
            for w in words:
                bag.append(1) if w in parent_words else bag.append(0)
            output_row = list(output_empty)
            output_row[classes.index(doc[1])] = 1
            training.append([bag, output_row])
        elif doc_counter == max_count: #If the maximum document has been reached, break
            break
    else:
        pos_counter += 1 #If the program has not yet reached the last processed document, continue to the next one
close_name = "trainingdump" + str(iteration) #Name of the saved file
pickle.dump( {'words':words, 'classes':classes, 'documents':documents,
'training':training, 'counter':doc_counter}, open( close_name, "wb" ) )

```

Running this code still caused a memory error. There are obviously too many intents for my computer to handle so even with splitting up the code into several modules and using pickled variables, the variables are too much for 16GB of memory dedicated to Python to handle thus, and unfortunately, I will have to decrease the number of intents being taught to the chatbot in order to get it to work. I know that there's no possible way for me to increase the amount of RAM to dedicate to Python (because the maximum amount is already dedicated) due to the fact that after the memory errors: any web pages that are open crash, the screen flashes black screens and some programs spasm/fail to work which would be due to lack of memory available.





Aw, Snap!

Something went wrong while displaying this webpage.

[Learn more](#)

(Some of the after effects on the computer when trying to use tabs in Chrome. Only a restart or waiting ~15 minutes would fix this.)

I can easily remove intents manually by deleting the last ~30,000 intents to make the number of classes be exactly 100,000. I will now test the original finished code with this number of intents to see if it makes a difference. I have named this edited file “intents2.json”.

There has been no difference with ~38,000 less intents so I will lower down the number of intents until I reach a point where there’s enough RAM to run the code without there being a memory error. Ideally, I would imagine that the chatbot would require at least 40,000 intents to be able to respond well enough to context. However, the more intents that I am able to process, the better the chatbot will be. To enable the program to process more intents, I should split this model into two (as previously proposed). The program would process all of the intents and pickle them. In a new module, these will be unpickled and used to create a neural network (which will hopefully not experience the same memory errors).

Now that I come to think of it, a possible solution is to split up the intents file into several smaller files, then create the numerical data for each of these to later combine them all together so they can be trained together in the model. The first step to do this is to figure out what is the maximum amount of intents that can be converted to numerical data and pickled at a time (which is what I was trying to do in the paragraph before). Note that due to the pickling, it could

decrease the amount of intents that can be processed at a time. The following code was run in a separate module with different lengths of intents to figure out what number of intents is the maximum.

```

from nltk.tokenize import word_tokenize
from nltk.stem.lancaster import LancasterStemmer
import json
import string
import numpy as np
import pickle

punc_dic = dict((ord(punct),None) for punct in string.punctuation)
words = []
classes = []
documents = []

with open('intents2.json') as json_data:
    intents = json.load(json_data)

for intent in intents['intents']:
    pattern = intent['parent']
    w = word_tokenize(pattern)
    words.extend(w)
    documents.append((w, intent['tag']))
    classes.append(intent['tag']) #An if statement check was removed because I realised that all the tags are unique so the next tag would never be in classes so a check is unnecessary

stemmer = LancasterStemmer()
words = [stemmer.stem(word.lower()) for word in words]
words = [word.translate(punc_dic) for word in words]
words = sorted(list(set(words)))
words.remove('')
#print(len(documents), "documents")
#print(len(classes), "tags", "classes")
#print(len(words), "unique stemmed words", words)
training = []
output_empty = [0] * len(classes)

for doc in documents:
    bag = []
    parent_words = doc[0]
    parent_words = [stemmer.stem(word.lower()) for word in parent_words]
    parent_words = [word.translate(punc_dic) for word in parent_words]
    while '' in parent_words: #I realised that after translating, any punctuation would be turned into '' so these should be removed from the list
        parent_words.remove('')
    for w in words:
        bag.append(1) if w in parent_words else bag.append(0)

    output_row = list(output_empty)
    output_row[classes.index(doc[1])] = 1
    training.append([bag, output_row])

training = np.array(training)
train_x = list(training[:,0])
train_y = list(training[:,1])
pickle.dump( {'words':words, 'classes':classes, 'train_x':train_x, 'train_y':train_y}, open( "training_data", "wb" ) )
#This is done sooner so it can be used by training part 2 and the next stage of code (using the model) as well
print("data pickled successfully")

```

Note that a major change made was the removal of the use of the “random” module. The line of code “random.shuffle(training)” before “training = np.array(training)” was removed to prevent a possible memory error. I did not think that shuffling the training data would have made a significant enough difference to the code so to prevent a possible error (because so much data would have to be shuffled), I removed it.

Using 48,000 intents, this module successfully turned the intents into numerical data, with the console printing out “data pickled successfully” and there being a formation of a file called “training_data”. Because there are ~138,000 intents and only 48,000 can work at a time, that means that I have to create two more Json files for three in total (intents2, intents3 and intents4). The file “intents3.json” will contain the next set of 48,000 intents and the file will be used in this code to create a file called “training_data2”. The file “intents4.json” will contain the last 42,000 intents and the file will be used by this code to create “training_data3”. The second intents file intents3, was also successfully pickled. The third intents file, intents4, was pickled successfully, and must be faster than the other two due to it being smaller.

Now that I have three “training_data” files, I can create the second module where the “train_x” and “train_y” data are combined together and then used to train a neural network. Hopefully, the training of the model using these three intents file will not cause any memory errors. The minimum amount of training that the chatbot should have is at least one of the intents files. Using an SQL database instead of pickling may allow me to teach the chatbot model more intents so that it could be more context accurate. However, I don’t want to delve into an unplanned feature so much so depending on how well the chatbot responds to context questions in the next stage of development, adding SQL to the project may be in its own, later development stage.

The following code was used for the second module for the creation of the model itself.

```

import pickle
import tensorflow as tf
import tflearn

training_data1 = pickle.load(open("training_data", "rb")) #Open the numerical data so
the train_x and train_y can be extracted
train_x1 = training_data1['train_x'] #Train_x of the first set
train_y1 = training_data1['train_y'] #Train_y of the first set
#training_data2 = pickle.load(open("training_data2", "rb")) #Open second set of
numerical data
#train_x2 = training_data2['train_x'] #Train_x of the second set
#train_y2 = training_data2['train_y'] #Train_y of the second set
#training_data3 = pickle.load(open("training_data3", "rb")) #Open third set of
numerical data
#train_x3 = training_data3['train_x'] #Train_x of the third set
#train_y3 = training_data3['train_y'] #Train_y of the third set

train_x = [] #Create the train_x list
train_y = [] #Create the train_y list
train_x.extend(train_x1) #Add the first train_x to the train_x list
#train_x.extend(train_x2) Add the second train_x to the train_x list
#train_x.extend(train_x3) Add the third train_x to the train_x list
train_y.extend(train_y1) #Add the first train_y to the train_y list
#train_y.extend(train_y2) Add the second train_y to the train_y list
#train_y.extend(train_y3) Add the third train_y to the train_y list

tf.reset_default_graph()
net = tflearn.input_data(shape=[None, len(train_x[0])])
net = tflearn.fully_connected(net, 8)
net = tflearn.fully_connected(net, 8)
net = tflearn.fully_connected(net, len(train_y[0]), activation='softmax')
net = tflearn.regression(net)

model = tflearn.DNN(net, tensorboard_dir='tflearn_logs')
model.fit(train_x, train_y, n_epoch=250, batch_size=200, show_metric=True) #It would
take too long to do 1000 epochs of so many intents so the number had to be lowered. The
batch size was increased for more accuracy and lower run times for training
model.save('model.tflearn')
print("model created successfully")

```

As can be seen in the code above, several sections have been coded up. This is because, when attempting to open more than one pickled data during the session (for example “training_data” and “training_data2”), there would be a memory error. This means that although I have tried to use the other ~90,000 intents from the pickles, they consume too much data to open (using pickle at least). If I find that the bot is lacking its accuracy with answering with

context in the next section, I can make some changes to the project to include SQL which may allow me to use more intents for training.

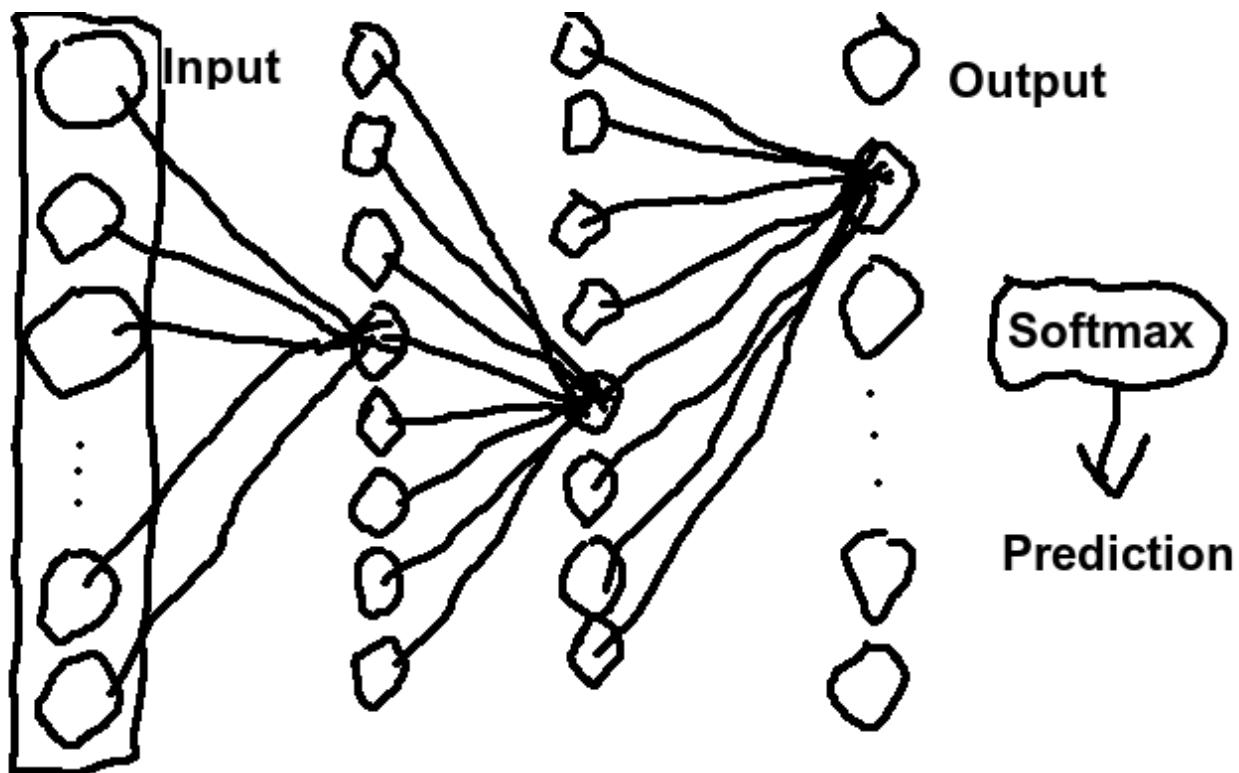
When this code ran, it took several hours to complete (because it only trained 200 intents at a time out of 48,000 which had to be done repeatedly for 250 times). When this code completed its run, there was an output of “model created successfully” which meant that the model was successfully created. The training took 60,000 steps with 250 epochs completed, a total loss of 0.95444 and accuracy of 0.8197. I am not sure of the units used for the losses, but considering that most of the numbers were above one, it’s very unlikely to be a percentage. Having an accuracy of 82% with the model is really good, it’s a high number for thousands of intents.

This table below shows what the variables from the code above look like, taken from the processing of intents3.

Name	Type	Size	Value
bag	list	13762	[0, 0, 0, 0, 0, 0, ...]
classes	list	48000	[48001, 48002, 48003, 48004 ...]
doc	tuple	2	(['Sleep', 'tight', '.'], 96000)
documents	list	48000	[(..., 48001), (...], 48002), (...], 48003), ...]
intent	dictionary	3	{'tag':96000, 'parent':'Sleep tight.', 'child': 'Don't let the bed bugs bite.'}
intents	dictionary	1	{'intents':[..., {...}, {...}, ...]}
output_empty	list	48000	[0, 0, 0, 0, 0, 0, ...]
output_row	list	48000	[0, 0, 0, 0, 0, 0, ...]
parent_words	list	2	['sleep', 'tight']
pattern	string	1	“Sleep tight.”
punc_dic	dictionary	32	{64:None, 124:None, 125:None, ...}

train_x	list	48000	[[0, 0, 0, 0, 0, ...], [0, 0, 0, 0, ...], ...]
train_y	list	48000	[[1, 0, 0, 0, 0, ...], [0, 1, 0, 0, ...], ...]
training	object	(48000, 2)	Array from numpy
words	list	13762	['0', '0300', '05', ...]

Below is a drawing of what this neural network looks like:



On the left side of the diagram, there are a bunch of units (more specifically, units in the length of the “words” list or bag of words as those are the input data). I couldn’t draw so many thousands of units so I just drew the first 3, a continuation and the final 2. This part represents the first layer created in the code, the input layer. The next layer created after this is the fully connected hidden layer which contains eight units. Each of the hidden layer units is connected to each unit from the input layer. This was represented on one of the units (it has been abstracted so it’s easier to see what’s going on in the diagram), however this type of connection would be seen for every single unit on this hidden layer. The next layer is another fully connected hidden layer with eight units. Each unit from the first hidden layer connects to

each unit in the second hidden layer, as shown in the diagram with one unit. The final layer is the output layer which has a “softmax” activation. This has units the length of the “classes” list or the bag of words for the classes. This has been represented by five circles for abstraction. All units run through a softmax activation function which will give a probability to each of these units. If the model thinks that a certain unit is the one wanted (or in other words, if that class is the class said by the user), then that unit will have the greatest probability. This probability forms part of the prediction, where the other part is the class which got that particular probability. When the model takes in a bag of words as the input, the output would be a class and a probability. The hidden layers are very important because they work to figure out what words represent which output unit. These do all the work and they are what figure out the most likely class said from the input. I may need to add more units to my hidden layer because there are so many intents if the current classification does not work properly (as in, the classes found by the model aren't accurate enough).

Stage 2 Review

What has been done?

All of the training data created in Stage 1 has been converted to numerical data and then pickled separately. A third of this numerical data was then used to create and train a neural network created by Tensorflow and TFlearn. This model takes in a bag of words and outputs the predicted class and the prediction of that class.

How has this been tested?

Throughout the development of this stage, I have used many debugging print statements to test out if the data structures were correct. For example, I checked whether or not “train_x” and “train_y” lists contained only 0’s and 1’s. This would have also been checked by Numpy as if there was a non-integer value in these lists, there would be an error from Numpy when converting the list to an array as an array can only store one data type. Another way in which debugging was used to test this is during the training of the model. There were many outputs into the console during training which contained information on the epoch number, the training step, the accuracy of what was learned during this step, total loss of data and more. This allowed me to adjust and correct my model. For example, when using a batch size of eight instead of 200, I saw that the total loss of data ranged between 9-12. On the other hand, a batch size of 200 provided a total loss of 1-11 (with losses of 10-11 in the first epoch number, the rest ranged in numbers but were always below 7).

Moreover, when deciding between stemming and lemmatization, I conducted a series of tests to compare the two. For example, the main test was comparing the length of the words list since making it as short as possible (without removing any unique words) would make the “train_x” longer when it doesn’t need to be, as the length was increased by non-unique words. This allowed me to choose the best possible stemmer which in turn, improved the outcome of the model.

Furthermore, numpy converted the bags in the lists into arrays which meant that the bags indeed only contained the numbers 1 and 0. This is important because Tensorflow only accepts inputs of 2D arrays, not 2D Python lists.

How does this meet with the success criteria and/or requirements?

This training stage is not in any of the requirements however it does relate to points on the success criteria. This stage meets the “Training of the chatbot” point in the success criteria. It is also related to “The bot is able to respond with a relevant reply” and “Use of machine learning

for responses". This is because the bot would not be able to respond with a relevant reply without the neural network being created first. As well as this, the main use of machine learning was used in this stage, when creating a neural network which will learn from the training data and provide the output needed to provide the user with contextualised responses. The loss of thousands of intents due to this stage does affect the success criteria "The bot is able to respond with a relevant reply" however 48,000 intents may actually be enough for context.

Changes in design due to this section

There were some additions to the design that occurred in this section. Everything to do with NLTK or any form of language manipulation (such as removing punctuation) was not originally in the design. This was added in this section to improve the model by making messages more likely to be equal to one another. Stemming removed any non-unique words from the "words" list. The removal of punctuation prevented it from having any effect on the model's predictions. The huge changes made were the splitting of the module into two, with code changes so that it made a suitable split. These were not planned because I did not know how many intents would be extracted from Cornell's Corpus prior to the development stage.

Summary of the project so far

So far, the project consists of a neural network which is the chatbot model. It is able to take in a bag of words to predict which class is the most similar to the input. The training data acts like a database from which the model learns from and uses to check what the suitable response for a certain message is. I have also outputted a file from the "pickle" library which stores the data structures that were created in this stage so that they can be used in another module.

Stage 3 Response Creation:

This next section of the project consists of using the trained model to predict data. Ideally I want to generate a response to any sentence the user types in. However, the model takes in a bag of words, not a string input. The model will generate a list of probabilities for all of the classes. To generate a response, the following must occur:

1. I will need to get some input from the user (which in the completed project would be from the user interface)
2. Convert the input into a bag of words
3. Use the model to create a prediction of this bag
4. Find the most probable class said
5. Use the child of that class from the intents file as a response

I will need to use the data structures created in the previous section of code. This is why, in the beginning of the previous section of code, a library called “Pickle” was imported. This library will allow me to save data structures which can then be imported and used here. I will also have to use the intents file (“intents2.json”) to extract and use the responses stored. To start off with, I will need to import all of the libraries used in this module. These will all be the same imports as in the previous stage, including an earlier import (which was later removed) random.

```
from nltk.tokenize import word_tokenize
from nltk.stem.lancaster import LancasterStemmer
import numpy as np
import tflearn
import tensorflow as tf
import random #For randomly picking suitable responses
import json
import pickle
import string

training_data = pickle.load(open("training_data", "rb"))
words = training_data['words']
classes = training_data['classes']
train_x = training_data['train_x']
train_y = training_data['train_y']
punc_dic = dict((ord(punct),None) for punct in string.punctuation)
stemmer = LancasterStemmer()

with open('intents2.json') as json_data:
    intents = json.load(json_data)
```

After loading the same imports, the numerical data is un-pickled. The intents file is also opened because I will need to extract the children to be used as the chatbot's response. The next thing that needs to be done is loading in the saved Tensorflow (TFlearn framework) model.

```
tf.reset_default_graph()
net = tflearn.input_data(shape=[None, len(train_x[0])])
net = tflearn.fully_connected(net, 8)
net = tflearn.fully_connected(net, 8)
net = tflearn.fully_connected(net, len(train_y[0]), activation='softmax')
net = tflearn.regression(net)

model = tflearn.DNN(net, tensorboard_dir='tflearn_logs')
model.load('./model.tflearn') #Load the saved model
```

In the code above, the Tensorflow model structure has to be redefined, just like how it was done in the previous stage. The model can then be loaded and used in this module.

As discussed at the beginning of this stage, there are several steps that must take place in order for there to be a response produced. The first step cannot occur here, unless with myself inserting testing strings, as it is reliant on the user's input from the user interface. This connection between the user interface and this module will be created in the next stage. The

second step is to convert the input into a bag of words which must occur before the program begins to process the intents. The technique of converting a user input into a bag of words will be very similar to the process done in the previous stage, so that the same format of data is produced that's suitable for processing. These are done by the two functions below:

```
def cleanUp(sentence): #Function for cleaning up the user input
    sentence_words = word_tokenize(sentence.lower()) #Tokenize the words and turn it
    all into lower case
    sentence_words = [word.translate(punc_dic) for word in sentence_words] #Removes all
    punctuation in the sentence by turning them into None or ''
    while '' in sentence_words: #To remove all cases of '', in scenarios where the
        sentence consists of more than 1 punctuation as .remove() only removes the first
        case found
        sentence_words.remove('')
    sentence_words = [stemmer.stem(word.lower()) for word in sentence_words] #Stem each
    word
    return sentence_words #Returns the "cleaned" words list

def bagOfWords(sentence, words): #Function for creating a bag of words
    sentence_words = cleanUp(sentence) #Clean the sentence
    bag = [0] * len(words) #Creates an empty bag of words
    for s in sentence_words: #For each word in the sentence
        for i,w in enumerate(words): #For each word in the word list, where w is the
            word and i is the position of that word
            if w == s: #If the words match
                bag[i] = 1 #Place a 1 in that position in the bag
    return(np.array(bag)) #Return a numpy array version of the bag of words
```

The “bagOfWords” function uses the “cleanUp” function to clean up a sentence (the user input) by: making it lowercase, removing punctuation and stemming the words. This makes it the exact same format as the words stored in the words list. Then, the bag of words is created by checking for where the word exists in the bag of words. If it exists, that position of the word is marked as a 1. When the model later reads this bag of words, it can see that the bag of words contains that particular word as it was marked with a 1. After using this function, a suitable bag of words has been created for the user’s input, which completes the second step.

The rest of the steps (three to five) can be done in one function, a response function. This function will use the model to predict the most probable class said. Then, I can figure out what class it is and where it is to find the “child” stored in the intents file, which can then be used as a response. In my design, I created a classification function which will be used to find the most probable class. Then, the response function will use this result to find the best suitable response. There may be several suitable responses which is why the Random library will be used to randomly select a response to use when there’s no longer a way to differentiate

between them. However, I found that there is an easier way of doing this in one function, a response function, using the Numpy library, without having to use Random or any Lambdas (like done so in the design section). This is done in the code below:

```
def response(sentence): #Function which takes in a user input
    results = model.predict([bagOfWords(sentence, words)]) #The sentence is converted
    into a bag of words. This bag of words is then inserted into the model to output a
    set of predictions for the most probable class
    results_index = np.argmax(results) #The index of the highest predicted class is found
    tag = classes[results_index] #The class/tag is found using the index

    for intent in intents['intents']: #For each intent
        if intent['tag'] == tag: #If the tag is equivalent to the most probable tag
            response = intent['child'] #Then its child is the response
            break #Breaking from the for loop because the tag was already found so I no
            longer have to iterate through it
    return response #Return the response found
```

Now it's time to test how this entire module works with responding to a message. In the previous section, I discussed adding SQL in order to possibly allow me to train the chatbot model with more intents. However, I wanted to test how well the chatbot works with context in order to judge whether it needs more training. Prior to testing, I created my own hypothesis stating that the longer a user input is, the more accurate the response to it would be. I think that the more words a message has, the more likely that a prediction containing important context words will be made. Some of the tests and the responses created to the tests can be seen below. These tests will be rated in colours, **correct in context**, **can be applicable** and **not suitable** by me.

“My friends want to talk to you.” -> “**Don’t be fooled by appearances.**”

“How are you?” -> “**Head still secure to the neck.**”

“What are you doing?” -> “**I need ten thousand dollars. My lawyer says if we can spread ten thousand bucks around, we can get the case dropped.**”

“My favourite colour is blue” -> “**Man named Turk Baker?**”

“What’s going on?” -> “**Mike, there’s something that’s been on my mind and I have to tell you about it.**”

“My name isn’t Mike...” -> “**That’s five clicks, Jackson**”

“Who is Mike and what do you have to tell me?” -> “**The poem?... The poem was great!**”*

“By the way, my name isn’t Jackson either haha” -> “**I figured that out.**”

“What’s it like being a chatting robot?” -> “**Yeah, she’s the girl who drove the bus in “Speed.” She’s cute.**”

“I miss playing video games all the time. Do you like playing video games too?” -> “**We are here to discuss our merger. Naturally I can make no statement at this time. We must begin at the beginning.**”

“Who is your favourite character in the Hunger Games? Mine would probably be Katniss Everdeen because she is awesome (and is an amazing archer!).” -> “**Really?**”

“I have to leave now, goodbye chatbot!” -> “**No.**”

“Why no? You don’t want me to leave?” -> “**We won’t. Let’s go inside.**”

“I have to go to sleep, good night!” -> “**We’re going right ahead with--*what*?**”

*(I was conflicted when considering this as a bad response or could be applicable. It doesn’t answer the first question about who Mike is however if you take “what do you have to tell me?”, “The poem? The poem was great!” could be a suitable response because it does actually answer my question)

Out of these 14 test strings, 11 were correct in context or could be interpreted as correct in context. This means that, using this very small sample, 79% of the responses respond correctly to context. Using more samples would create a more accurate percentage rate of correct responses created using this model. Assuming that the model actually gets 70-75% of the messages correct in context, it’s a fairly good rating for a conversational chatbot. My hypothesis also seems to be false, the chatbot seems to be equally able and unable to respond to both long and short sentences, with none being over the other.

The following is a table of what the variables used in this module look like:

Name	Type	Size	Value
classes	list	48000	[1,2,3,4,5,6,7,8,9...]
intents	dictionary	1	{"intents": [{"..."}, {"..."}, ...]}

punc_dic	dictionary	32	{64:None, 124:None, 125:None, ...}
train_x	list	48000	[[0,0,0,0,...], [0,0,0,0,...], ...]
train_y	list	48000	[[1,0,0,0,...], [0,1,0,0,...], ...]
training_data	dictionary	4	{'words':[...], 'train_y':[...], ...}
words	list	14098	['0', '00983', '0200', ...]

Stage 3 Review

What has been done?

The chatbot model can now be used to predict suitable responses for the best possible contextually correct answer. The model predicts the likelihood of each class it has been taught has been said, then the most predicted class is found. When this class is found, it's searched for in the intents file where, when found, the child message can be extracted so that it can be used as the response.

How has this been tested?

I conducted a series of tests to check whether SQL will be used in the project in the aim of providing more intents to teach to the model. These tests consisted of me randomly typing different messages to the model which were formatted, converted into bag of words and then processed to figure out the best possible response. I used debug “print” statements to run the “response” function and print out the chatbot’s response to each of my messages which were then documented. Each one of the responses were rated according to how accurate of a response they are. They each received a rating out of three: accurate, can be interpreted as accurate and not accurate. Half of the tests resulted in an accurate response. Three of the tests provided a not accurate response while the rest were categorised as “can be interpreted as accurate”.

How does this meet with the success criteria and/or requirements?

There are two requirements from the success criteria which have been met here: “The bot is able to respond with a relevant reply” and “Use of machine learning for responses”. The chatbot model uses deep learning to figure out the context said in the message and replies with a possible suitable response.

Changes in design due to this section

One of the changes was the removal of one of the subroutines. In the design section, I checked to see if the predictions were greater than a certain number then randomly picked one of the classes which fit the criteria to use its child as the response. To find these possible classes, I used the classify subroutine. The response subroutine in the design then picked out the response to use according to the classes found by the classify subroutine. However, in this section, I have removed the classify subroutine completely and adapted the response subroutine accordingly. This is because I believe that it’s better for the model to find the most

suitable response to the user rather than a lot of possibly suitable responses to randomly pick from.

This first change led to another change, the removal of the use of the Random library in this project. Initially, it was meant to be used in this stage for randomly picking one of the possible suitable responses in the response subroutine but because it changed to find the most suitable response instead, there was no longer any use for random.

Summary of the project so far

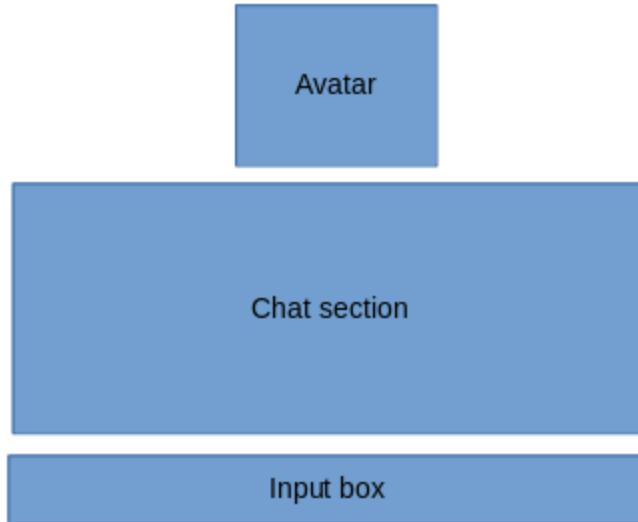
So far, this project consists of a json file acting as a response database and a neural network model that uses machine learning to determine the best response for a given string input. The model uses the response database to pick out the responses.

Analysis of this stage and the previous stage

The fact that not all of the responses were accurate to context does not concern me, no conversational chatbot is perfect, there are occasions where the neural network gets something wrong such as predicting a different class as the more probable class compared to another, more suitable one. However, I do expect for the chatbot to be correct a lot of the time, maybe even most of the time. If the chatbot was trained using more intents, it would be more context accurate. However, the downside effect of having too many intents is that the run-time of this module is greater due to: increased intents file thus it takes longer to read through and find the correct most probable class; greater size of “words” list so more words to search through; greater sizes of bags of words and a longer training process. All of these points except the last one may have a great effect on the time it takes for the chatbot to respond to the user while in the user interface. I want the response to seem immediate but with a huge amount of intents, it could take several minutes for the chatbot to respond, which is not ideal (especially since this is supposed to be used in a conversation). Also, the training process for 48,000 intents took around 13 hours to complete (as there were 250 repeats of 48,000 data and only 200 data were processed at a time). If the batch size was increased to greater than 200, the training process would have occurred faster however I did find that when using a batch size of 1000 caused memory errors so I wanted to play it safe using 200. Decreasing the number of repeats would also increase the training speed but at the loss of accuracy. With a much greater intents file, this may venture into days or even cause a MemoryError Exception during the training process, even with very low epochs and high batch sizes. Moreover, having more intents but less epochs defeats the point as yes, the model will have access to more data so it can recognise more contexts but it will lack the training to do that so it may not predict properly. Therefore, using SQL instead of pickling may not fix this problem.

Stage 4: Main User Interface

The following is the design that I came up with in the design section:



I will be using Tkinter to handle the user interface, including the main menu, buttons and text objects. I have made a basic main menu which represents the design above. However, I added an extra feature which is not part of the initial design which is an exit button. Below is the code that I have used to create this menu.

```
from tkinter import * #Imports the Tkinter library

root = Tk() #Creates a Tkinter window
root.title("Chatbot") #Makes the title of the window be "Chatbot"

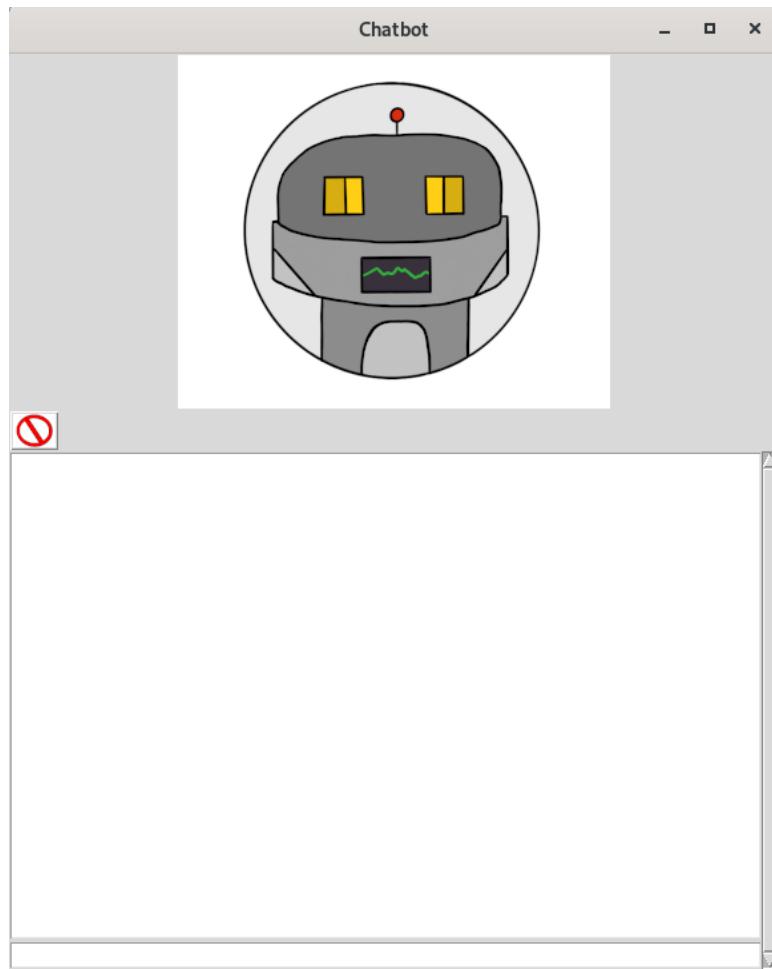
img = PhotoImage(file = "avatar.png") #Creates an image using the avatar image file
Label(image = img).pack() #Turns the image into a label and adds it to the window

exit_photo = PhotoImage(file = "leave.png") #Creates an image using the leave image file
leave = Button(root, image = exit_photo) #Creates a button in the window with an image
in it
leave.pack(side = TOP, anchor = W) #Adds the button into the window on the top left side

scroll = Scrollbar(root) #Creates a scrollbar
scroll.pack(side = RIGHT, fill = Y) #Adds the scrollbar to the window on the right side,
filling it up
messages = Text(root, wrap = NONE, yscrollcommand = scroll.set) #Creates a text box
with the scrollbar
input_user = StringVar() #Creates a Tkinter string variable
input_field = Entry(root, text = input_user) #Creates the input field where the text
inputted is the string variable created above
messages.pack() #Adds the message box to the window
scroll.config(command = messages.yview) #Makes the scrollbar able to view content on
the y axis so it works
input_field.pack(side = BOTTOM, fill = X) #Adds the input field to the window at the
bottom and fills up the entire section on the x axis

frame = Frame(root) #Creates a frame for the window
frame.pack() #Packs the frame so it can be seen
root.mainloop() #Runs the user interface
```

This is the output of my code:



In the code, I first import the necessary libraries. So far, the user interface only requires Tkinter but it does use several libraries in Tkinter. The user interface will also rely on another module created by me which consists of the response creation and conversion to numerical data called "Model.py". This will have to be imported later on so that I can send over the message entered by the user, analyse this message in Model.py and create a response then send it back to the user interface so that it will be displayed for the user. When the user interface has been completed, a link between these two modules will be made.

The layout of the user interface is very similar to the design. The main difference between the user interface and the design is the added exit button. This was added to allow the user to stop the program whenever they want during their session. Currently, none of the elements on the user interface actually work (other than technically the scrollbar but there's nothing to scroll through). This is because I have not assigned any code telling them what to do. This will be my next step. There's also a scroll bar in the chat section which will allow the user to scroll through the conversation during their session. This has been created as a Tkinter scrollbar element then

assigned to the chat section using “messaged = Text(root, wrap = NONE, yscrollcommand = scroll.set)” and “scroll.config(command = messages.yview)”.

This is the new code which adds onto their functionality:

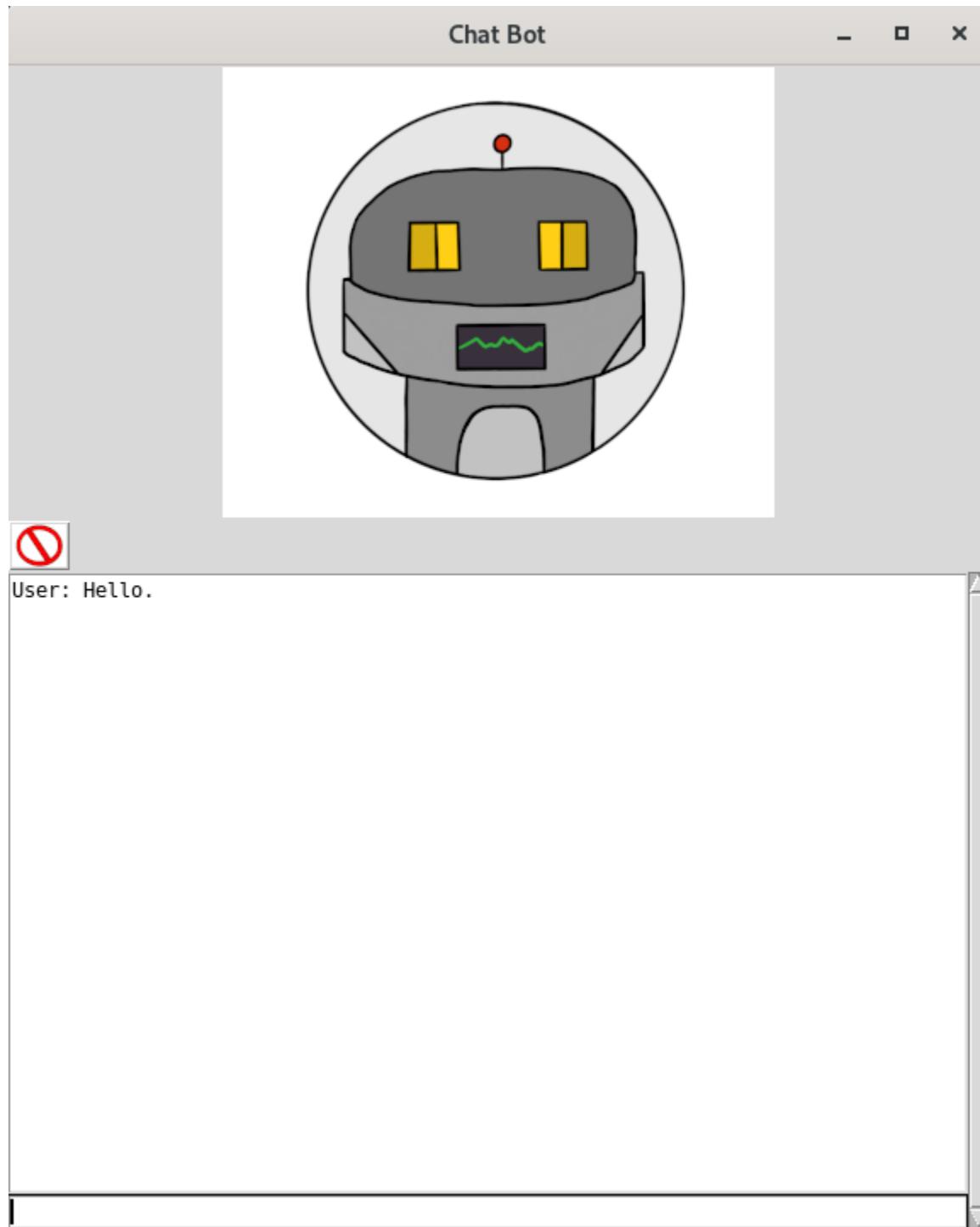
```
def enterPressed(event): #Function that is run when the user presses enter in the input field
    input_get = input_field.get() #Gets the input said by the user
    print(input_get) #Prints this input to test if it worked correctly
    messages.insert(INSERT, '%s\n' % ("User: " + input_get)) #Inserts a message into the message box on a new line formatted as: "User: *their message*"

    input_user.set('') #Sets the input box to blank
    return "break" #Returns that the function has finished

def stopProgram(event): #Function that runs when the exit button has been pressed
    root.destroy() #Closes the window

leave.bind('<Leave>', stopProgram) #Binds the stopProgram function to the leave button so it runs when pressed
frame = Frame(root)
input_field.bind("<Return>", enterPressed) #Binds the enterPressed function to the input box so when enter is pressed, this function will run
```

This code added new features to the user interface, the functionality features for entering a message and for the exit button. I have added two new functions, “enterPressed”, and “stopProgram”. The “enterPressed” function gets the user’s message by taking the current input from the input field. This will then be printed into the console for debugging purposes (to make sure that the program is gathering the correct message). Then, using “messages.insert(INSERT, ‘%s\n’ % (“User:” + input_get))”, the program will print out the user’s message like this:

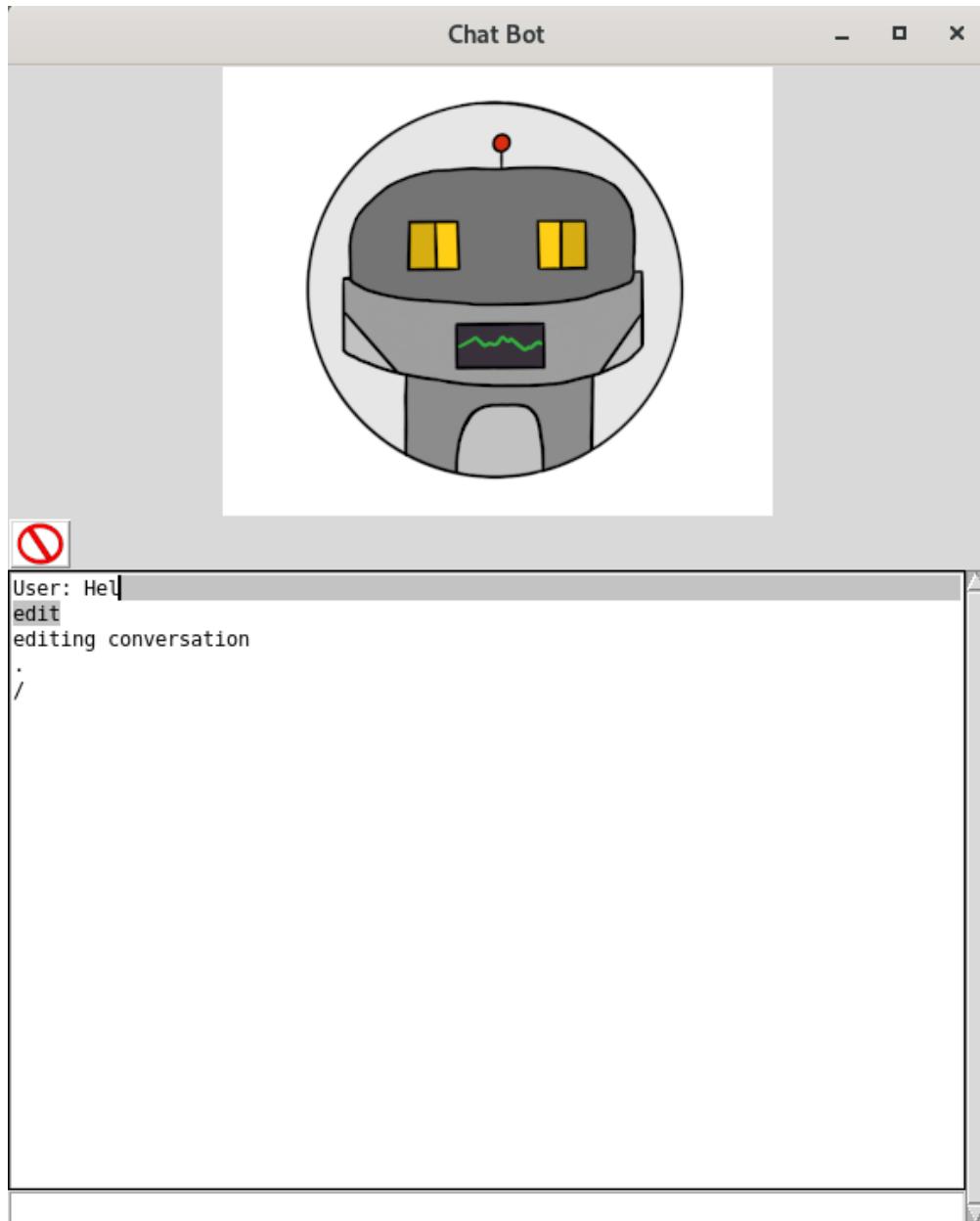


As seen above, the message was correctly displayed in the message box and the console outputted "Hello.".

Next, the input box gets cleared, ready for the next user input. Finally, the 'return "break"' stops raising the event when it's no longer needed. The "stopProgram" function is very simple; when it

runs, it causes the program to terminate. These functions are bound to their respective element, “stopProgram” being bound to the exit button so it will run when clicked and “enterPressed” bound to the input box when the enter key is clicked.

A problem that I found with the user interface is that the user is able to edit the chat section. This defeats the point of having a chat section as the user can delete any message from there or even add new characters.



To prevent this, I have to disable the use of that specific Tkinter Text element, with this being the “messages” variable. To do this, I can simply use “messages.config(state = DISABLED)”

after creating the Text element. However, a problem that I found with this is that the chat section can no longer be edited after it has been disabled thus the messages to be displayed can't be displayed. To fix this, I change the state of the chat section to normal when it needs to be edited and then, after adding the message, it will be disabled. This process happens so fast that it prevents the user from being able to edit the chat section when the state allows them.

The following is the new code containing the changes of state:

```
messages.config(state = DISABLED) #Disables editing
input_user = StringVar()
input_field = Entry(root, text = input_user)
messages.pack()
scroll.config(command = messages.yview)
input_field.pack(side = BOTTOM, fill = X)

def enterPressed(event):
    input_get = input_field.get()
    #print(input_get)
    messages.config(state = NORMAL) #The messages box can now be edited so the line below works
    messages.insert(INSERT, '%s\n' % ("User: " + input_get))
    messages.config(state = DISABLED) #The box is disabled again, quickly enough so that the user will not be able to edit it when in NORMAL
```

After doing some more research, I decided to change the style which my code was written to replace my use of the procedural style. This new style uses object oriented programming in Python by using classes. There are multiple benefits of using object oriented programming for my user interface such as: it is giving me a private namespace for all of my callbacks and private functions as well as generally making it easier to organise my code. In the procedural style, I had to code top-down, having to define functions before using them (such as defining the “enterPressed” function before use) which meant that I had to create the main menu at the very end, as seen from my earlier user interface.

Furthermore, instead of importing each library that I use separately, like earlier, here I decided to import only the libraries that I will be using from Tkinter. The problem with “from tkinter import *” is that I import many libraries from Tkinter that I don’t end up using which wastes data. Also, this may result in conflicts when using methods as some libraries may contain methods with the same names as others. In order to avoid these problems, I later changed my code from “from tkinter import *” to “import tkinter as tk”. This will allow me to use specific Tkinter functions and methods by specifying “tk.[name of function/method]”. This solution prevents me from having global namespace pollution in my code. Each item in the user interface uses at least two lines

of code, one for the creation, where information such as linked images is specified, and the packing which specifies where it is placed in the window.

Another problem that I found is with the button; it technically does its job, but not precisely. When the exit button is clicked, the button itself and all other elements on the window are frozen which disables the user from being able to continue talking to the chat bot (or using anything else in the user interface). However, the window itself doesn't close, so while the program itself has terminated, the frozen window remains so the user would have to click the default exit button as well to close this window (this default button is found on any windowed application on their computer). When I ran the same code on a different computer, the program did terminate completely when clicking on the exit button so this is unlikely to be a bug with the code.

All of the fixes discussed above can be seen in the code below:

```
import tkinter as tk #More efficient method of importing Tkinter as I only import and
use what I need

class Application(tk.Frame): #Defines a class called Application which takes in a
Tkinter frame
    def __init__(self, name): #Reserved method in python classes.
        tk.Frame.__init__(self) #Initialises the class
        self.name = name #Creates a variable to set the name of the window
        self.name.title("Chat Bot") #Using the variable above, the name is changed

        self.exit_photo = tk.PhotoImage(file = "leave.png")
        self.leave = tk.Button(self, image=self.exit_photo, command=self.quit) #I found
        that self.quit does the exact same thing as the earlier stopProgram function
        self.leave.pack(side = tk.TOP, anchor = tk.W)

        self.img = tk.PhotoImage(file = "avatar.png")
        tk.Label(image = self.img).pack()

        self.scroll = tk.Scrollbar(self)
        self.scroll.pack(side=tk.RIGHT, fill=tk.Y)
        self.messages = tk.Text(self, wrap=tk.NONE, yscrollcommand=self.scroll.set)
        self.messages.config(state=tk.DISABLED)
        self.input_user = tk.StringVar()
        self.input_field = tk.Entry(self, text=self.input_user)
        self.input_field.bind("<Return>", self.enterPressed)
        self.messages.pack()
        self.scroll.config(command=self.messages.yview)
        self.input_field.pack(side=tk.BOTTOM, fill=tk.X)

    def enterPressed(self, par1):
        self.input_get = self.input_field.get()
        self.messages.config(state=tk.NORMAL)
        self.messages.insert(tk.INSERT, '%s\n' % ("User: " + self.input_get))
        self.input_user.set('')
        self.messages.config(state=tk.DISABLED)
        return "break"

if __name__ == "__main__": #Executes the user interface only if it's the main file
being executed (as in, it's not imported and executed in another file). This is
important because the user interface will use "Model.py".
    root = tk.Tk()
    Application(root).pack() #Packs the window created so it can be seen
    root.mainloop()
```

Later in this stage, the user interface will be linked to the “Model.py” module to collect the response created. This means that the user interface file would have to run before, so it can be seen, therefore it acts like a main file. To make sure that it is indeed acting like a main file (and not imported then executed in another file) I check if the name is equal to “`__main__`”. If the name is equal to “UserInterface” then it means that this module has been imported into another because of the line “`import UserInterface`”. An unmentioned addition to the code above is “`par1`” in the “`enterPressed`” function. This was added to prevent an exception for two variables used for the function but only one is defined.

This is the main part of the user interface done. Its main functionality is complete because the user is able to enter messages and “receive” responses (the connection to the chatbot model will be done in the end). The next section of code involves extra features requested by the stakeholders during their interview.

With the leisure side of the chatbot complete, as a user can now have a conversation with the chatbot, I have to add more educational features to fit the requirements for the stakeholder. My stakeholder has suggested features such as saving and loading conversations. This will be implemented (as well as other small features to aid the educational side of the bot) in the user interface. I have created a menu bar and added three sections to it: File, Help and Settings. The following is the code for File:

```
self.menu = tk.Menu(self.name) #Creates a menu
self.name.config(menu = self.menu) #Adds the menu to the application, forming a menu bar
self.file = tk.Menu(self.menu) #Creates another menu
self.menu.add_cascade(label = "File", menu = self.file) #Adds the menu to the menu bar,
naming it "File"
self.file.add_command(label = "New", command = self.restart) #Adds a tab to the File
menu called "New". When this tab is clicked, it runs the function called "restart"
(which is easily accessed thanks to the object-oriented programming structure of this
code)
self.file.add_command(label = "Load", command = self.load) #Adds a tab to the File menu
called "Load". When this tab is clicked, it runs the function called "load"
self.file.add_command(label = "Save", command = self.saveText) #Adds a tab called
"Save" which calls the function "saveText" when clicked
self.file.add_command(label = "Exit", command = self.quit) #Adds a tab called "Exit"
which terminates the program
```

To make the code used in the functions worked, I added these imports to the list of imports:

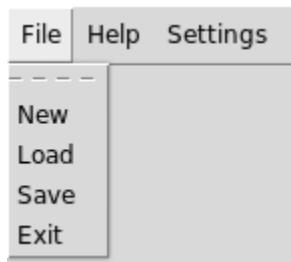
```
from tkinter.simpledialog import askstring #Prompt window which gets user input
from tkinter import filedialog #Prompt window for selecting files

def restart(self): #Function for restarting the chat section
    self.messages.config(state=tk.NORMAL)
    self.messages.delete('1.0', tk.END) #Deletes all of the contents
    self.messages.config(state=tk.DISABLED)

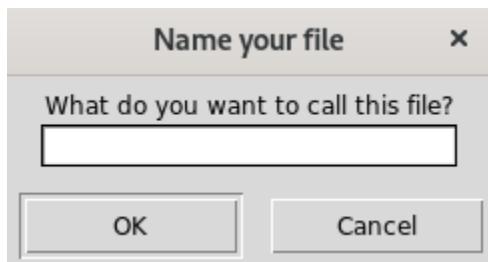
def load(self): #Function for loading a saved conversation file
    self.file_name = filedialog.askopenfilename(initialdir = "./", title = "Select
    conversation file to load", filetypes = (("Text files", "*.txt"),("All files", "*.*")))
    ) #Opens a dialog for file selection. The directory location of the file is what this
    variable stores
    self.file_name = self.file_name.split('/') #Splits the directory into a list of names
    of the folders
    self.file_length = len(self.file_name) #Gets the length of the amount of folders
    there are
    self.file_name = self.file_name[self.file_length-1] #The file name is the last item
    of the list e.g. "conversation.txt"
    with open(self.file_name, 'r') as file: #Opens the file selected
        self.conversation = file.readlines() #Reads all of the lines in that conversation
        and stores them
        self.messages.config(state = tk.NORMAL)
        self.messages.delete('1.0', tk.END) #Deletes all of the contents of the message box
        (so it clears the message box in case there was a previous conversation)
        for line in self.conversation: #For each line in the loaded conversation
            line = line.strip('\n') #Format the line so there's no "\n" after each line
            self.messages.insert(tk.INSERT, '%s\n' % (line)) #Insert the line on a new line
        self.messages.config(state = tk.DISABLED)

def saveText(self): #Function for saving a conversation in a file
    self.file_name = askstring('Name your file', 'What do you want to call this file?')
    #Opens a dialog which takes in a string
    with open(self.file_name+'.txt', 'w') as file: #Creates the file as a text file
        self.conversation = self.messages.get("1.0", tk.END) #Reads the entire message box
        for its contents
        file.write(self.conversation) #Inserts the ocntents of the conversation into the
        file
```

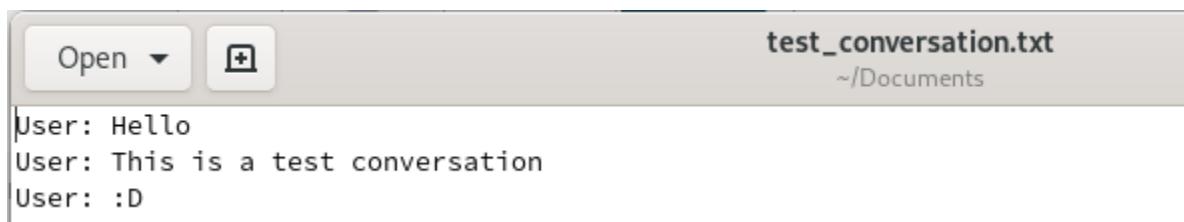
Which adds the following into the user interface:



When clicking “Save”, it does this:

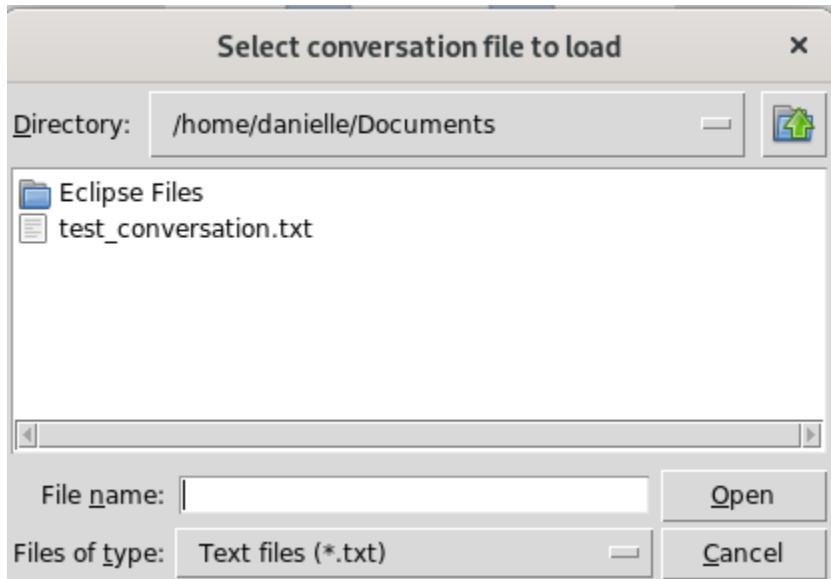


Entering a file name creates a text file which stores the conversation as shown in the example below.



The code “(“1.0”, tk.END)” is seen several times in the code, in the restart, load and save functions. What this code does is select the entirety of the text in the chat section. The first part, “1.0”, means that the input should be read from line one, character zero. The “tk.END” part means to read until the end of the text box thus the entire text has been selected. This allows me to get the entire conversation and insert it into the text file.

When clicking “Load”, it does this:



This allows a user to load a previously saved conversation that they had with the chatbot. Selecting a file name refreshes the chat section and adds the conversation from the file into the chat section.

```
User: Hello
User: This is a test conversation
User: :D
```

Here, I am saving the text file in the same directory as the one the python script is stored in. This allows the script and text files to always be stored together (unless the user moves the text file post saving). In my computer, this directory is “/home/danielle/Documents”. The directory has ‘/’ which split up the folders (this format is the same for many different types of computers). The file itself will always be the last value in the list so I can get the name of the last value in the new list, after splitting the address by the character ‘/’. Then, I can open the file, reset the chat section (so inserting new lines don’t conflict with the old lines) and insert all of the lines in the conversation file.

Clicking “New” simply refreshes the chat section. It deletes all of the text inside that text box. Technically, this isn’t a new session however the chatbot model doesn’t work in sessions, like a user might. It won’t be determining what it’s writing/responding to next according to what it responded before thus for the user, deleting all the text looks like it’s a new session. The chatbot doesn’t need to work on a new session to act like it’s in a new conversation because it doesn’t use the message history so there’s no point in refreshing the model as well.

The “Exit” option does the exact same thing as the exit button, it terminates the program.

There are several problems that I found with some of this code. In the “Load” file screen, if the user clicks escape/exits the menu without selecting a file, there is an error because none type cannot be split into a list (as I do when splitting the directory). Therefore, I need to create some kind of error message when the user exits the selection menu. This leads on to the minor problem with the “Save” section. If the user exits the screen or clicks save with no input, the program will search for a file named “ ” and then return an error in the console. To prevent this, I need to check if the user input is a “None” type and use a message box to show an error message.

To fix the problem when the user leaves the file screen without selecting a file, I have added the code below. This also required some new imports which have been added to the import list

```
from tkinter import messagebox

def load(self):
    self.file_name = filedialog.askopenfilename(initialdir = "./", title = "Select
    conversation file to load", filetypes = (("Text files", "*.txt"),("All files", "*.*"))
)
    if self.file_name == None or self.file_name == '' or self.file_name == ' ': #If a
        file wasn't selected
        return #Then return to prevent errors
    try: #Try to run this line of code
        self.file_name = self.file_name.split('/')
        self.file_length = len(self.file_name)
        self.file_name = self.file_name[self.file_length-1]
        with open(self.file_name, 'r') as file:
            self.conversation = file.readlines()
            self.messages.config(state = tk.NORMAL)
            self.messages.delete('1.0', tk.END)
            for line in self.conversation:
                line = line.strip('\n')
                self.messages.insert(tk.INSERT, '%s\n' % (line))
            self.messages.config(state = tk.DISABLED)
    except: #If there's an error for whatever reason
        messagebox.showerror("File Error", "There was an error gathering your file, please
        try again.") #Show an error message
```

I found that when the user closes the file selection dialog or clicks cancel, it returns an empty value. Therefore, I check if the file’s name is empty to check if they simply closed the dialog. The use of “try” and “except” allows the program to run the code as usual but if something goes

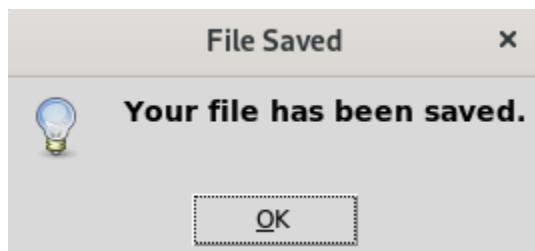
wrong (and there's an error), an error dialog box will be displayed, instead of the program printing errors into the console.

A similar problem occurs in the “Save” menu when you do a similar thing, leave the menu without entering a user input. The code below solves this.

```
def saveText(self):
    self.file_name = askstring('Name your file', 'What do you want to call this file?')
    if self.file_name == None or self.file_name == '' or self.file_name == ' ':
        return
    try:
        with open(self.file_name+'.txt', 'w') as file:
            self.conversation = self.messages.get("1.0", tk.END)
            file.write(self.conversation)
    except:
        messagebox.showerror("Saving Error", "There was an error with saving your file,
        please try again.")
```

When the user exits the dialog box using the cancel button or the x button, the variable “self.file_name” is set to null/none. This is because no input was entered. Therefore, to prevent an error occurring when the user wants to exit the dialog box without saving a conversation, it just checks if the file name is empty, if it is then it ignores the saving process by exiting before it occurs. I have also added the “try” and “except” to allow the use of an error message in case something does go wrong during this process. I also added a message box to show that the file has been saved after saving (as no message after could confuse the user on whether the file saved successfully or not).

```
messagebox.showinfo("File Saved", "Your file has been saved.")
```



The second section is the help section. The code used for this can be seen below.

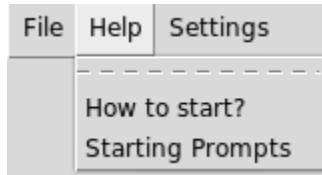
```
self.help = tk.Menu(self.menu)

self.help.add_command(label = "How to start?", command = self.startBox)
self.help.add_command(label = "Starting Prompts", command = self.promptBox)

def startBox(self):
    messagebox.showinfo("How to start?", "To start, simply type in the input box (the
smallest box/the bottom box) a message and the bot will respond!") #Shows the info in
a message box

def promptBox(self):
    messagebox.showinfo("Starting Prompts", "Suggested prompts:\n 'Hey!'\n 'How are you?
\n 'Are you real?'\n 'Can I be your friend?'")
```

The output of this code looks like this:



Clicking on "How to start?" or "Starting Prompts" will open a message dialog as seen below:



With the settings menu, there are two key things that the user is going to be able to edit: whether text-to-speech is enabled and whether or not to use slow mode. Both of these options are for text-to-speech. Before I add this feature into the project, I have coded the functions and submenus in which these features will be used.

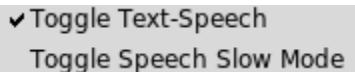
```
self.text_to_speech = tk.IntVar() #Creates a Tkinter integer variables
self.text_to_speech.set(0) #On default, text to speech is disabled. 0 = clear, 1 =
tick so a clear value is text-to-speech disabled
self.slow_mode = tk.IntVar() #Another Tkinter integer variable
self.slow_mode.set(0) #n default, slow mode is disabled (because text-to-speech is off)
self.settings_menu = tk.Menu(self.menu)
self.settings_menu.add_checkbutton(label = "Toggle Text-Speech", onvalue = 1, offvalue
= 0, variable = self.text_to_speech, command = self.toggleSpeech) #This creates a
submenu with a checkbutton
self.settings_menu.add_checkbutton(label="Toggle Speech Slow Mode", onvalue = 1,
offvalue = 0, variable = self.slow_mode, command = self.toggleSlow)
self.menu.add_cascade(label = "Settings", menu = self.settings_menu)
```

The settings for the checkbutton “onvalue” sets the value when it’s checked is a 1 and the “offvalue” sets the value when it’s off as a 0. This is used for the “self.text_to_speech” and “self.slow_mode” to check whether these have been enabled later on. When these submenus are clicked, the functions “toggleSpeech” and “toggleSlow” will be called accordingly.

```
def toggleSpeech(self): #Function for toggling text-to-speech
    enabled = self.text_to_speech.get() #Gets the value of the checkbox
    try: #In case something goes wrong, my program can display an error message
        if enabled == 1: #If it has been checked
            messagebox.showinfo("Text-Speech", "Text-Speech has been enabled!")
            #Tell the user that it has been enabled
        elif enabled == 0: #Else, if it hasn't been checked
            messagebox.showinfo("Text-Speech", "Text-Speech has been disabled!")
            #Tell the user that they have just disabled text-to-speech
    except:
        messagebox.showerror("Text-Speech Error", "There has been an error. Your setting did not take affect.")

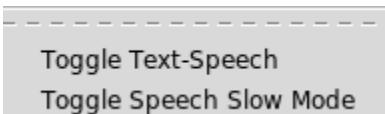
def toggleSlow(self): #Function for toggling slow mode
    enabled = self.slow_mode.get() #Gets the value of the checkbox
    try:
        if enabled == 1:
            messagebox.showinfo("Slow Mode", "Slow Mode for Text-Speech has been enabled!")
        elif enabled == 0:
            messagebox.showinfo("Slow Mode", "Slow Mode for Text-Speech has been disabled!")
    except:
        messagebox.showerror("Slow Mode Error", "There has been an error. Your setting did not take affect.")
```

On the user interface, this code shows up as:

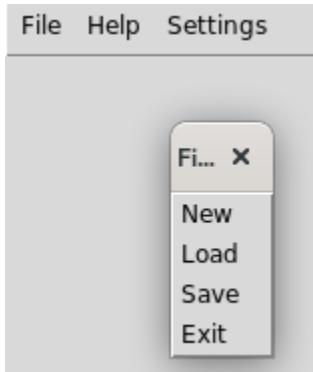


The “Toggle Text-Speech” submenu has been checked meaning that text to speech has been enabled. When clicking on any of these subroutines, the application will show dialog boxes, telling the user what has been done (similar to the information dialog boxes from earlier on).

When running the user interface, there's a feature found which I did not like.



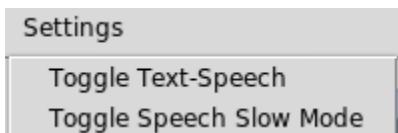
First of all, the dashed lines do not look aesthetically pleasing at all. They are present in all of the submenus and are there due to the “tear off” feature in the menu bar. This is demonstrated below:



When one of the menus on the menu bar is dragged off, it creates its own new window with the same, working submenus. Using one of these “ripped off” windows work exactly the same as using them on the main window. However, I find that it can get a bit annoying, having so many open windows at the same time and I also didn’t like the look of the dashes provided due to this feature so I added in code to remove this.

```
self.option_add('*tearOff', tk.FALSE) #Prevents the ability to "tear off" menus
```

Which now results in the submenus looking like this:



That’s exactly how I wanted the submenus to look like and there’s no longer a problem with dragging off the menus into separate windows.

Before adding the text-to-speech into the user interface, I want to link the user interface to the “Model.py” module created in the previous stage. This is because the text-to-speech will be used on the chatbot’s responses however I need some kind of responses for it to be used on. To link the user interface module with “Model.py”, I will be importing it using the “Importlib” library and then use the response function from that module to create the chatbot’s response. The first thing that needs to be done is importing the new library.

```
import importlib
```

Then, I need to import the module itself which can be done with the following line of code:

```
self.bot = importlib.import_module("Model") #Imports Model.py into the user interface module
```

Now that “Model.py” has been imported into this module, it can be used. I will only need to use the model to generate responses when the user has inputted a message, otherwise there would be no response to send over for analysis by the model. But, the user only inputs a message when they have pressed the “enter” key on the input box. This event is checked by the “enterPressed” function, so that’s where the responses have to be created.

```
self.response = self.bot.response(self.input_get) #Sends over the input written by the user to the response function in Model.py  
self.messages.insert(tk.INSERT, '%s\n' % ("Chatbot: " + self.response)) #Insert the message said by the chatbot into the user interface in a similar way to how the user's message has been inserted
```

Whilst running this code, I greeted the chatbot to see if it would respond to test this new code. When greeting the chatbot, this was the response:

User: Hello there!
Chatbot: Greetings.

Because the chatbot has responded to my message and it was successfully displayed, I can tell that this linkage between the modules has been successful thus it works.

Another feature to add before text-to-speech is the spell checking. Before the model should be used to create the responses, I wanted to spell check the user and to notify them if they have misspelled a word. This is needed for the educational side as in this way, a person can learn how to spell words that they often misspell. This would also make saving conversations more useful as any newly found misspelled words can be documented so they can focus more on them in the future. To handle the spell checking side of the chatbot, I will be using the library “PyEnchant”. I will first start off with adding this library to the list of imported libraries.

```
import enchant  
import random
```

When using the enchant library to spell check a word, it results in a list of word[s] of suggestions. These suggestions are the possible word[s] that the user meant to use but misspelled them. However, in the suggestion list, there could be more than one suggested word therefore there must be some way of picking one out to use as a suggestion. That's the role of the “Random” library, it will randomly pick one of the suggestions to use. Now, I need to set up the spell checking itself. This code should be done in a separate function that's located outside of this class because it would only run when called. I will call this function “spellChecker”.

```
ench = enchant.Dict("en_GB") #Uses an English dictionary to spell check the messages  
  
def spellChecker(words): #Function for checking if a word has been spelled correctly  
    suggestion = [] #A list of possible words that were misspelled  
    for word in words: #For each word in the list of words  
        if ench.check(word) == False: #If the word has been misspelled  
            suggestion = ench.suggest(word) #Suggests a list of word[s] that are close  
            to this one  
            picked_suggestion = suggestion[randint(0, len(suggestion)-1)] #Randomly  
            picks a suggestion  
            response = "Did you mean " + picked_suggestion + " instead of " + word + "?  
            #The response to use  
        else: #If there's no incorrectly spelled words  
            #Run the response function
```

The way that this function works is that it takes in a list of words (the user input after being tokenized). Then it will use the English enchant dictionary to check whether each word in the word list is found inside. If it's not, then the word has been misspelled. If it has been misspelled, I use enchant to suggest a list of words that this misspelling correlates to - which word has the user misspelled? I randomly pick out one of the words using the “Random” library (if the list has

one word then it will select that one word) then set a response message with that picked suggestion. The else is where the response function would then come in. I will only need to use the “randint” function from the “Random” library so I will be importing the library like so: “from random import randint”.

There are two problems that I realised with this code, firstly, there’s no value being returned if spell checking has occurred. This can easily be fixed by adding a return statement like “return response”. The second problem is that the function needs to take in a tokenized list of words however there’s no NLTK imported or used here to tokenize the words here. There’s also no other modules where this function can be stored instead because, for example, “Model.py” is for all of the functions done by the model itself (and spell checking is not). But, if I tokenize the words here, when they are inserted into the “response” function in “Model.py”, they will go through another tokenization, which may cause errors. Instead, what I could do to fix this is to have this function take in a string (the user input) then tokenize the words to check for any spelling errors. If there are no spelling errors, then the string is passed on to the “response” function. This can be seen done below.

```
ench = enchant.Dict("en_GB") #Uses an English dictionary to spell check the messages

def spellChecker(sentence): #Takes in a string, not necessarily always a sentence
    suggestion = []
    words = word_tokenize(sentence) #Tokenize the user input
    for word in words:
        if ench.check(word) == False:
            suggestion = ench.suggest(word)
            picked_suggestion = suggestion[randint(0, len(suggestion)-1)]
            response = "Did you mean " + picked_suggestion + " instead of " + word + "?"
        else:
            response = bot.response(sentence) #Creates the model's response
    return response
```

If I add this function into the “`__init__`” function and the “application” class, I won’t need to input any variables to the function to either “enterPressed” (to display the response) or to this function. With some adjustments made to this code, I can add it into the class.

```
self.ench = enchant.Dict("en_GB")

def spellChecker(self):
    self.suggestion = []
    self.words = word_tokenize(self.input_get)
    for word in self.words:
        if self.ench.check(word) == False:
            self.suggestion = self.ench.suggest(word)
            self.picked_suggestion = self.suggestion[randint(0, len(self.suggestion)-1)]
            response = "Did you mean " + self.picked_suggestion + " instead of " + word
            + "?"
        else:
            response = self.bot.response(self.input_get)
    return response
```

With some further adjustments made to the “enterPressed” function, this spelling feature should work.

```
self.response = self.spellChecker() #Runs the spell checking and possibly the model's response creation and returns the response
```

Now running the code and entering sentences with correctly spelled words will output the model’s response. Entering incorrectly spelled words will output a response “from” the bot, which possibly corrects the user. There are a few problems that I found with this. The first one being, when I misspelled “hello” several times, the program outputted once “help”, another time “halo” and a third time “hello”. This shows me that the spell checking does not always predict correctly which word was misspelled. This means that it may suggest a correction that doesn’t in fact correct what they wrote (for example, suggesting a new word that the user knows how to spell). The second problem is that punctuation isn’t removed so the spell checker corrects the user when they use punctuation, with examples shown below.

```
User: testing spell checking [like so]
Chatbot: Did you mean a instead of ]?
User: Using punctuation! :)
Chatbot: Did you mean o instead of )?
```

This is not ideal. The user isn’t making a spelling mistake but they are still being corrected. I can fix this by removing punctuation before spell checking but I would rather not do any language manipulation on the text in the user interface module as it’s not directly related to the user interface. The final problem is that in the success criteria, the program is supposed to be giving the user responses using machine learning. However, the spell checking responses are not done using machine learning and the response to the user’s input will completely skip over

the context and will only correct the spelling mistake. On the other hand, the model itself will likely ignore the misspelled word as it's not in the bag of words and still provide the best possible response that it can. Thus, I will be removing the spell checking feature from this program.

The final feature for the user interface is the text-to-speech. This will be easy to implement. This feature is provided by the Python library “pyttsx3”. Firstly, it will have to be imported into the project using “import pyttsx3”. Then, the text-to-speech engine has to be loaded in which is done by this line of code:

```
self.engine = pyttsx3.init() #The text-to-speech engine
```

The next step is using the text-to-speech itself. This is done by the following lines of code which make use of the loaded text-to-speech engine.

```
self.engine.say(self.response) #Tells the engine what to say (which is the chatbot's response)
self.engine.runAndWait() #Says the text out loud
```

In the menus created earlier in this stage, I added the “settings” menu. In the “settings” menu, I created two submenus to toggle text-to-speech and slow mode. They have no functionality so clicking check on either of the boxes doesn’t do anything yet, it needs to be implemented. This will be the next step.

```
if self.text_to_speech == 1: #If text-to-speech has been enabled
    if self.slow_mode == 1: #If slow mode has been enabled
        self.engine.setProperty('rate', 125) #Slows down the rate at which words are set
        self.engine.say(self.response)
        self.engine.runAndWait()
    else: #If slow mode is not enabled
        self.engine.setProperty('rate', 200) #Sets the rate back to default (if it isn't already)
        self.engine.say(self.response)
        self.engine.runAndWait()
```

With text-to-speech working as well as slow mode, the user interface has been completed.

Stage 4 Review

What has been done?

Using Tkinter, I have created the main user interface which is where all of the interactions with the user will be made. This user interface is very similar to the design in that it contains the avatar, the chat section and the user response section. An exit button has also been added. There is a menu bar containing a lot of submenus. They give options for the user to save and load conversations as well as edit the settings for text-to-speech options. They have all been linked to functions and they work properly. The user is able to insert their input onto the chat section without being able to edit the chat section themselves.

How has this been tested?

For each part added to the user interface, I ran the program to see if it worked. I also tested if each function worked for example by typing into the user input and seeing if it's displayed in the correct place. One of the tests that was documented above was finding out that the user was able to edit the chat section - deleting and adding messages to several lines. There are no changes to be done for the main user interface as it all works however I may want to insert more buttons or menus to increase the features that the user interface has to improve the user's experience. Another test was entering user inputs to check if the bot has responded using responses created in "Model.py" and checking if the spell checking works.

How does this meet with the success criteria and/or requirements?

The user interface meets a lot of requirements listed in the success criteria.

For the design section, the user interface meets all of the following requirements:

- Simple design
- Conversation feed with scroll bar
- An obvious user input section
- An avatar for the bot

These are all of the requirements listed in the design requirements.

The following are all of the requirements met in the functionality section:

- A user input section
- A submit function
- Some form of message box

- Avatar
- Main menu/screen

All of the requirements for the functionality section are met.

In the success criteria itself, the following are all of the requirements that this user interface meets:

- Simple design
- Option to press a button to stop the program
- Clear message box which shows the chat between the user and the bot
- Text area for input
- The bot is able to respond with a relevant reply

Changes in design due to this section

The user interface has mostly followed the design so far. The avatar, the chat section and the user input section have all followed the design. A minor difference was the addition of the exit button. There has also been the addition of submenus to allow the user to change different aspects of the user interface which was one of the major changes. Another major change was the removal of the spell checking functionality from the project. It was a planned feature but it conflicted with the machine learning success criteria and also provided several unnecessary problems to the project.

Summary of the project so far

This was the final step of the project, all the features have been added and the entire success criteria has been met. There's a user interface where the user can type to a chatbot that responds contextually back to the user. They can save, refresh and load conversations on the user interface. They can enable and change some of the settings on the user interface.

Stage 5: Final Testing

In the first part of the final testing, I want to check the user interface for any obvious mistakes, for example any typos, grammar mistakes or any type of usability errors. There were no problems found. The user interface does not contain a large amount of text other than anything written in the message boxes by the user or chatbot. The avatar image is centred, the submenus are all capitalised and no spelling mistakes were found.

The next step is to test any buttons or inputs into the program. These all include the submenus, the exit button and pressing enter in the input box. Starting off with the exit button, clicking the exit button terminates the program fully, with no errors seen in the console. Pressing enter into the user interface inserts any string written in the input box, even any blank strings, which is then displayed on the user interface. After it's displayed, the chatbot's response is displayed under this message. There were no errors seen and the user input box was refreshed. For each of the submenus, I tested its function. Under the "File" menu, I clicked on the "New" button which simply refreshed the message box by deleting all messages inside. The next submenu clicked was "Load", it opened a file selection menu and upon selecting a file, the text file was loaded in and presented in the user interface. Clicking cancel on this file selection menu did nothing to the user interface and did not represent any errors console errors. When I opened this file selection dialog and clicked on the "open" button without selecting a file, an error dialog was displayed. This was one of the error dialogs that I coded in the development section using a "try...except", in case the user does something wrong. This was an expected outcome because I intentionally did not select a file to see how the program would react.



When clicking on "Save" a string input dialog opens where the name of the conversation file to be saved can be created. I created a file called "text_conversation" and when loading it back into the program, it remained exactly the same. If the file has been saved successfully, another dialog appears to inform the user. Clicking on the "Exit" submenu has the same function as clicking on the "Exit" button, it terminates the program. No errors were seen in console and the "File"'s submenus functioned correctly.

The next submenu test is for the submenus under “Help”. Each one of these should display an information dialog. These were opened then closed with no errors.

Finally, I need to test the submenus used in “Settings”. Here I need to test if I receive the correct dialog for enabling and disabling the text-to-speech and slow mode. When clicking on one of these submenus, an “enabled!” information dialog is displayed and a check mark appears next to the words in the submenu without any errors.

Now that all the features, when correctly used, seem to be working, it’s time for the destructive testing. I created a table in the design section on how I will destructively test the data coming in from the input box (the user’s messages). I expect the data inputted to be any string which can vary in size. If the data inserted is not something that the bot can analyse, I expect the model to still use some kind of response or to not respond at all.

Input	Valid/Invalid	Result
A-Z alphabet (in capitals)	Valid	Input works
a-z alphabet (in lowercase)	Valid	Input works
Word	Valid	Input works
She said, “Hello!”	Valid	Input works
£10	Valid	Input works
*	Invalid	There’s still a response from the bot, as expected
@	Invalid	There’s still a response from the bot, as expected
	invalid	There’s still a response from the bot, as expected

I did not design any destructive testing for the submenus in the design section so I designed the following tables to run these tests for each submenu.

“New” sub menu

Input	Valid/Invalid	Result
Messages in the text box	Valid	Empty message box, as

		expected
Empty message box	Valid	Empty message box, as expected

The new sub menu behaves the same way, whether or not there are messages in the message box. As shown above, these work as expected.

“Load” sub menu

input	Valid/Invalid	Result
Open a selected text file	Valid	Conversation loaded precisely into the message box with success dialog
Close window	Valid	Message box remains the same
Click open without selecting a file	Invalid	Error dialog message, as expected

The load sub menu has three possible outputs: selecting a file, closing the window or opening a file before selection. The third one has to give the user an error because they can't open an unselected file, the program doesn't know which file they want open. As seen by the tables above, this works as expected.

“Save” sub menu

Input	Valid/Invalid	Result
Strings such as “conversation”	Valid	Text file called “conversation” created with success dialog
Integers such as “123”	Valid	Text file called “123” created with success dialog
Characters such as “*&^”	Valid	Text file called “*&^” created with success dialog
None	Invalid	Window closes, no success dialog, as expected (no error was coded in for this)
Spaces like “ ”	Invalid	Window closes, no success dialog, as expected

Cancel	Valid	Window closes
--------	-------	---------------

The save sub menu can take in many different types of inputs as it has the user write in an input. Creating a file with no name or a blank name will cause the file creation to cease and simply return (as coded).

“Toggle Text-Speech” sub menu

Input	Valid/Invalid	Result
Checked	Valid	Submenu is checked, text-to-speech functions now enabled with a success dialog
Unchecked	Valid	Submenu is unchecked, text-to-speech functions disabled with a success dialog

“Toggle Slow Mode” sub menu

Input	Valid/Invalid	Result
Checked	Valid	Submenu is checked, slow mode now enabled with a success dialog
Unchecked	Valid	Submenu is unchecked, slow mode disabled with a success dialog

Both of these toggle sub menus have two different modes - checked or unchecked. This is because they are tick boxes. Both of these are valid inputs which handle different things such as whether the user wants the chatbot’s messages to be read back to them.

There are no more data inputs from the user. All of the possible types of inputs have been tested and validated. They all work as planned, mostly by running the function, closing the window or giving the user an error message because they have done something wrong.

Hardware

I tested this software on three different devices. Two of them follow all of the hardware requirements from the specification and one does not. The first one is a Windows 10 computer,

the code works perfectly with no errors, as intended. The second one is a Fedora Linux computer which, when it follows the specification, also works as intended. The third case was another Fedora computer however in this case, not all of the specifications were followed. More specifically, this laptop did not run in a virtual environment. This caused errors with Tensorflow and TFlearn as Tensorflow could not run. This shows that as long as this code runs in a virtual environment, there should be no errors.

Checklist

Using the checklist that I have created in the design section, I have gone through in this section each function thoroughly and checked the functionality of each part of the program. Below is the checklist that I created with completed sections:

What Is Being Tested?	Does It Work?
Check all text for formatting errors	Y
Exit button closes the application	Y
The user can't modify the chat section	Y
Scroll bar works to see message history	Y
Saving a conversation into a file	Y
Loading a conversation file	Y
Creating a new conversation in the same session	Y
User is able to enter messages in chat	Y
Bot can respond to messages in chat	Y
Bot can respond using machine learning	Y

Now that I am confident that there are no bugs and all the features work as intended, I can pass the code over to the stakeholders who will conduct further testing. They will then provide me with some feedback on their experience with the program.

Stage 6: Stakeholder Testing

For the stakeholder testing, I will give each stakeholder a copy of the software along with the specification instructions (to ensure that the program works on their devices) and provide them with a series of questions for them to fill out.

The following are the questions that I asked:

How fast did you start conversing with the chatbot?

Did you use the conversation file features?

Did you use the text-to-speech feature?

How useful were the help menus?

How was your conversation with the chatbot?

How would you rate the accuracy of the chatbot's responses to yours?

I have received the following responses from my stakeholders.

Gabriella:

How fast did you start conversing with the chatbot?

"I talked to the chatbot pretty quickly. I found that it was laid out easily enough for me to find my own way to start the conversation. I saw a clear separation between two boxes and since I couldn't type anything in the big box, I assumed that the smaller box was where I could type in. Without a submit button, I simply clicked enter every time I typed in a message and the chatbot responded."

Did you use the conversation file features?

"I did actually. I found it cool that I could make a new conversation all of a sudden then switch out to my old one, edit that then save it again. It was great fun. Some of the chat bot's responses were funny so I am very happy that I could record it. The chat bot also used some words which I didn't know from before so I saved those too."

Did you use the text-to-speech feature?

"At first, I didn't see that it was a feature because I straight away started chatting to the bot. But later, when looking at what else I can do, I enabled the text-to-speech to try it out. It was good because it can pronounce any words that I otherwise wouldn't know how to."

How useful were the help menus?

"I skipped passing them and didn't really use them because I found the user interface to be fairly simple enough so there was no need."

How was your conversation with the chatbot?

"It was really funny. It said some very weird things in response to my messages which I found funny. Some of the responses I could tell were dramatised, so it was fun to get a dramatic reply."

How would you rate the accuracy of the chatbot's responses to yours?

"The chatbot wasn't always accurate, sometimes it started talking about completely random things like when I asked for its favourite sport, it went on about some person named 'Blake'. However, I did notice that most of the time it did respond accurately so I do think that it's done its job."

Ella:

How fast did you start conversing with the chatbot?

"It didn't take me too long to start, when I saw the help menu, I wanted to read it before starting to make sure that I understood anything, in case there was anything complex. It was pretty easy to understand how the user interface works and how to chat with the bot so after reading I did get started pretty quickly."

Did you use the conversation file features?

"No. I found no use for creating and saving conversation files. Maybe I could have used it to save some of the funny responses that I have received."

Did you use the text-to-speech feature?

"I found it very useful. After some time chatting with the chatbot, I got tired of reading so I just let the text-to-speech do the work for me. I left it on normal mode because I found that slow mode was too slow for me, I could listen better to normal mode."

How useful were the help menus?

"Very useful. They helped me get started because I liked to know what's happening when I used an application without reading a huge amount of tutorials or having to play around with it due to the complexity."

How was your conversation with the chatbot?

"It was lovely because it always responded to whatever I said. Of course, not entirely accurately but I didn't expect it to always respond in the same way as a user because it is just a program. Some of the off-topic remarks I found were quite useful because I could continue the conversation that way, in case I didn't know what else to say."

How would you rate the accuracy of the chatbot's responses to yours?

"I found it accurate. I always used the messages that didn't reply to mine as conversation starters to always continue the conversation which I found fun as we could chat for a long time like this. The randomness also made it exciting because I didn't always know how the chatbot would respond."

Review

My stakeholders seemed to have enjoyed playing around and chatting with the chatbot. They each liked different features of the chatbot but overall, they appreciated the simplicity of starting a conversation and talking to the chatbot. There was a complaint about slow mode being a little too slow but the main complaint was that the chatbot was not always accurate with responses. On the other hand, they went on to say that having unpredictable responses made their experience much more enjoyable. This is because they don't always know how the chatbot will respond to their message and because these responses come from movies, they do tend to be dramatic. A simple solution for the slow mode is to create a variable selection window like a number scale where they can always change the rate at which the chatbot speaks to whatever number they wish to (at a given range). The wild responses are a much harder issue. It could only be fixed in a revised and future version of this software as there are many different ways of possibly fixing the memory problem with adding whole new sections to the code.

Evaluation

Criteria Met

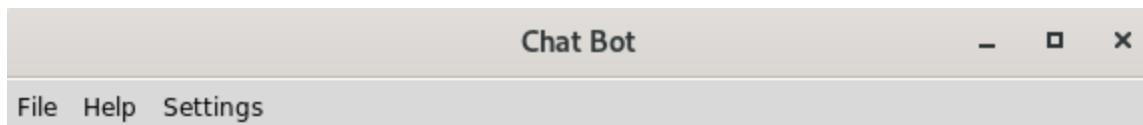
Criteria	Met?
Simple design	Yes
Option to press a button to stop the program	Yes
Clear message box which shows the chat between the user and the bot	Yes
Text area for input	Yes
Bot avatar	Yes
The bot is able to respond with a relevant reply	Yes (according to results from later survey)
Training data creation	Yes
Training of the chatbot	Yes
Use of machine learning for responses	Yes

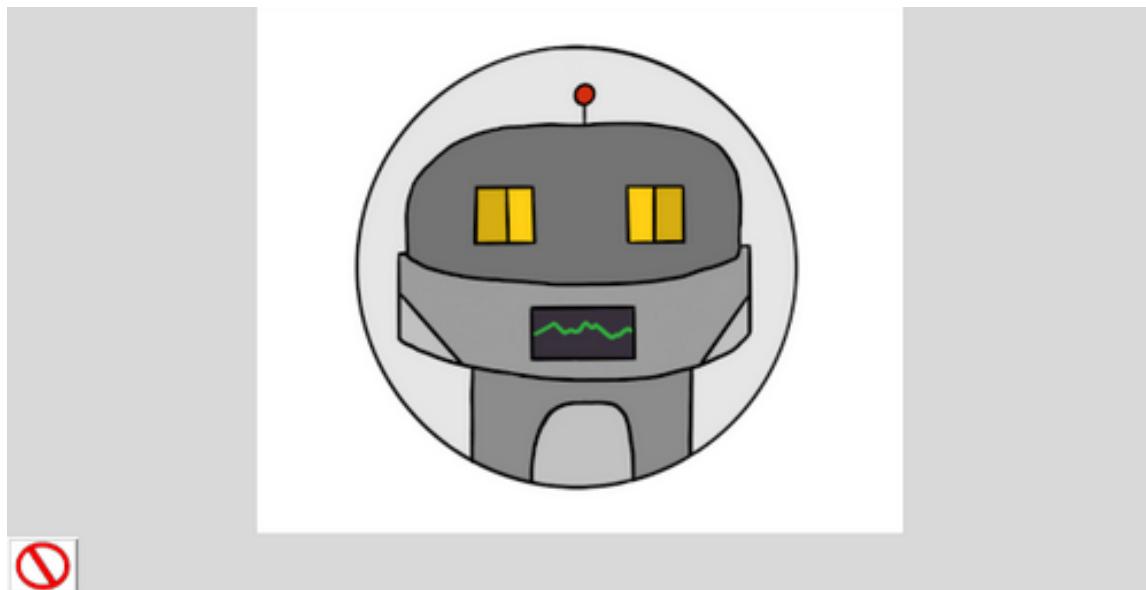
Spell checking was an early concept of the chatbot however, it was later removed as an objective so it was not included in this success criteria. I experimented with spell checking on a chatbot but found that it took too much from the aspect of the conversation (as there would be an automated message sent instead of responding to the user's message). I found that adding spell checking to the project would break the whole conversation idea that the stakeholders have suggested they wanted in their interviews therefore it was removed.

Evidence

Simple design:

Bot avatar:





Note: this image was split up to fit the page.

Option to press a button to stop the program:



Pressing this button causes the program to terminate. Pressing on the “Exit” submenu in the “File” menu (on the menu bar) does the same thing.

```
self.exit_photo = tk.PhotoImage(file = "leave.png")
self.leave = tk.Button(self, image = self.exit_photo, command = self.quit)
self.leave.pack(side = tk.TOP, anchor = tk.W)
```

Simple design (the simplicity test):

Person 1:

Name: Joe

Time taken to understand User Interface: 1 minute and 38 seconds

Complexity rating?: Very simple with only a few buttons and things to do which made it easy to navigate

Feedback: Took a while to find the text box because it was only a thin strip at the bottom.

Maybe make it bigger and a different colour or with an outline and something saying “type here...” which disappears when you start typing. Everything else was easy to use.

Person 2:

Name: Jay Rosenthal

Time taken to understand User Interface: 2 minutes and 13 seconds

Complexity rating?: Medium

Feedback: Mostly very simple. Features are easy to use and layout is simplistic. Help option in the taskbar was very useful. Textbox at the bottom could be larger or a different colour to make it clearer, as it took me a while to work out how to type. You could be prompted to see the help section at the start. Otherwise very good.

Person 3:

Name: Danny

Time taken to understand User Interface: 1 minute 37 seconds

Complexity rating?: Simple

Feedback: Nice and simple design and it is easy to use. But it isn’t clear where the input field is (although it is stated in the help section). Maybe you could move the input field above the output box to make it more clear and you could add a placeholder too. Other than that, it looks very good.

Person 4:

Name: Emily

Time taken to understand User Interface: 1 minute and 56 seconds

Complexity rating?: Fairly easy

Feedback: It’s a really simple design and it is easy to use. I struggled with finding the input field but after reading the help section, I easily found it. Maybe it could be made more obvious or bigger?

Person 5:

Name: Mia

Time taken to understand User Interface: 48 seconds

Complexity rating?: Simple

Feedback: I think the design is really simple. I immediately found the message box and started talking to the chatbot. Later, I played around with some of the other options like the text to speech.

Person 6:

Name: Amber

Time taken to understand User Interface: 1 minute and 15 seconds

Complexity rating?: Easy

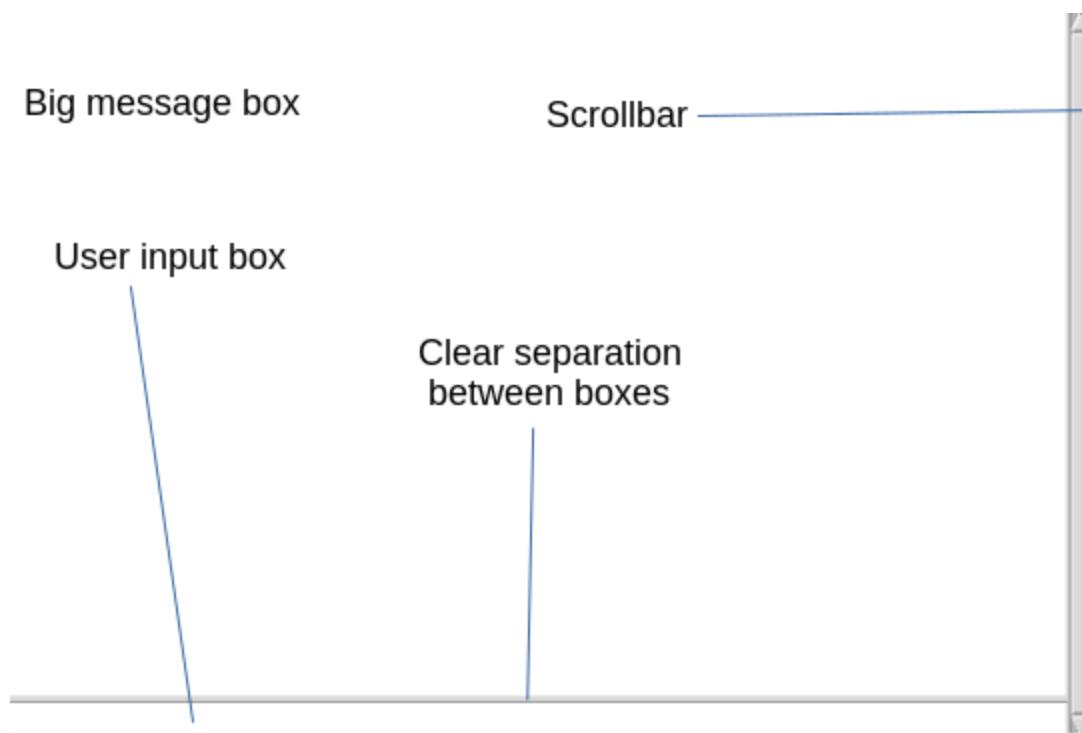
Feedback: Not hard to use so it is a simple design. I didn't understand what the new and open tabs meant so it took some playing around with them to see what happens. You could include a small tutorial to show a new user what each thing does however I do think it's currently pretty good.

Analysis:

The user interface seems to me mostly successful. Although some of the people who tested the user interface found it to be hard to find the input box, they did overall state that the user interface is simple in their feedback. As well as this, overall, their timings were not long, ranging from 48 seconds to 2 minutes and 13 seconds which are not long periods of times. Some found the help section to be useful and used them. There were a lot of good suggestions which could be implemented in an update to the program. Due to this feedback, the user interface was judged as simple so it has succeeded the requirement.

Clear message box which shows the chat between the user and the bot:

Text area for input:



User: Hello there!
Chatbot: Greetings.

Code showing how the user input is used and then represented.

```
def enterPressed(self, par1):
    self.input_get = self.input_field.get()
    self.messages.config(state = tk.NORMAL)
    self.messages.insert(tk.INSERT, '%s\n' % ("User: " + self.input_get))
    self.input_user.set('')
    self.response = self.bot.response(self.input_get)
    self.messages.insert(tk.INSERT, '%s\n' % ("Chatbot: " + self.response))
    self.messages.config(state = tk.DISABLED)
    if self.text_to_speech == 1:
        if self.slow_mode == 1:
            self.engine.setProperty('rate', 125)
            self.engine.say(self.response)
            self.engine.runAndWait()
        else:
            self.engine.setProperty('rate', 200)
            self.engine.say(self.response)
            self.engine.runAndWait()

    return "break"
```

The bot is able to respond with a relevant reply:

Example of the chatbot responding with a relevant response:

User: Hello there!
Chatbot: Greetings.

Code used for responses:

```
def response(sentence): #Function which takes in a user input
    results = model.predict([bagOfWords(sentence, words)]) #The sentence is converted
    into a bag of words. This bag of words is then inserted into the model to output a
    set of predictions for the most probable class
    results_index = np.argmax(results) #The index of the highest predicted class is found
    tag = classes[results_index] #The class/tag is found using the index

    for intent in intents['intents']: #For each intent
        if intent['tag'] == tag: #If the tag is equivalent to the most probable tag
            response = intent['child'] #Then its child is the response
            break #Breaking from the for loop because the tag was already found so I no
            longer have to iterate through it
    return response #Return the response found
```

Training data creation:

The code used for creating the training data:

```

def fileInsertion(parent, child):
    global tag #By setting tag as a global variable, this function can access the
    variable and edit it
    tag += 1 #Tag increments. Since its first value is 0, this sets it to 1 so the
    first paired messages will have the tag number 1
    data['intents'].append({
        'parent': parent,
        'child': child,
        'tags': tag
    }) #New tag value in the dictionary

def pairing(row): #The pairing function, used to be called dataFormatting
    conversation_texts = [] #This array will store all of the strings said in a conversation
    position = -1 #This variable marks the index of the current line when using the 2nd for loop below. It starts at -1 because the variable
    will increment until it's the last value so at -1, the first index will be 0
    new_list = row.split(' +++$+++' ) #This new list consists of: [character1, character2, movieID, [lineID, lineID, ...]] the line IDs vary
    according to how many lines have been said in this conversation
    if len(new_list) > 4: #The list has to have 4 pieces of data only, the 2 characters who are speaking to each other, the movie which this
    conversation is from and a list of the lines exchanged in this conversation
        return #If the current row isn't stored in the correct format e.g. an extra value because of corruption inside the file, I don't want
        to use this conversation
    lines = new_list[3] #The lines variable holds the list of line IDs from the conversation
    if len(lines) == 1: #If there is only 1 line said in this conversation, there's no pairing so we don't want this line is not a parent or a
    child
        return #So we move on to the next row in the file
    else: #If the conversation does have at least a parent and a child
        line_ids = lines.split(',') #The line ids are formatted like a list but it's actually a string so it needs conversion into a list so I
        can iterate through all of the IDs
        for line_id in line_ids: #For each ID in the list of line IDs said in this conversation
            position += 1 #The position value increments to mark the index of the line ID. This will allow me to determine if the line is a
            parent or a child
            formatted_id = line_id.replace("[", "").replace("]", "").replace("'", "").replace(" ", "") #The ids are compromised by array
            symbols (because it was an array stored as a string) so they must be removed in order to compare IDs in "movie_lines.txt"
            for line_data in movie_lines: #Searching through the movie lines corpus
                if formatted_id == line_data[0]: #Check to see if the IDs match
                    if position == 0: #Check if this is the first id in the list this doesn't have a parent but it is the parent of the next
                    message
                        conversation_texts.append(line_data[4].replace("\n", "")) #Appends the first message, it's the first parent so I will
                        need this data for the 2nd message which is the child
                    elif position == len(line_ids)-1: #If this is the last id in the list this can't be the parent of another message it can
                    only be a child
                        child = line_data[4] #Setting this line as the child
                        parent = conversation_texts[position-1] #Finding the parent from the conversation (it's always the message before)
                        #fileInsertion(parent, child) Inserting into the file
                        #print("parent:" + parent + "child:" + child) For debugging
                        #print(conversation_texts)
                    else: #A value that passes to this else statement is a value that isn't 1st or last which means that it acts as a child for
                    the value before and the parent of the next value
                        conversation_texts.append(line_data[4].replace("\n", "")) #As a parent for the next value, it must be appended so it
                        can be found by the next child
                        child = line_data[4].replace("\n", "") #Now it acts as the child
                        parent = conversation_texts[position-1] #The parent is the value before
                        #fileInsertion(parent, child) Inserting into the file
                        #print("parent:" + parent + "child:" + child) For debugging
                        #print(conversation_texts)

```

How the intents are stored in the JSON file:

```
{  
  "intents": [  
    {  
      "tag": 1,  
      "parent": "Can we make this quick? Roxanne Korrine and Andrew Barrett are  
        having an incredibly horrendous public break- up on the quad. Again.",  
      "child": "Well, I thought we'd start with pronunciation, if that's okay with  
        you."  
    },  
    {  
      "tag": 2,  
      "parent": "Well, I thought we'd start with pronunciation, if that's okay with  
        you.",  
      "child": "Not the hacking and gagging and spitting part. Please."  
    },  
  ]  
}
```

Training of the chatbot:

Code for training the chatbot:

```
import pickle
import tensorflow as tf
import tflearn

training_data1 = pickle.load(open("training_data", "rb")) #Open the numerical data so
the train_x and train_y can be extracted
train_x1 = training_data1['train_x'] #Train_x of the first set
train_y1 = training_data1['train_y'] #Train_y of the first set
#training_data2 = pickle.load(open("training_data2", "rb")) #Open second set of
numerical data
#train_x2 = training_data2['train_x'] #Train_x of the second set
#train_y2 = training_data2['train_y'] #Train_y of the second set
#training_data3 = pickle.load(open("training_data3", "rb")) #Open third set of
numerical data
#train_x3 = training_data3['train_x'] #Train_x of the third set
#train_y3 = training_data3['train_y'] #Train_y of the third set

train_x = [] #Create the train_x list
train_y = [] #Create the train_y list
train_x.extend(train_x1) #Add the first train_x to the train_x list
#train_x.extend(train_x2) Add the second train_x to the train_x list
#train_x.extend(train_x3) Add the third train_x to the train_x list
train_y.extend(train_y1) #Add the first train_y to the train_y list
#train_y.extend(train_y2) Add the second train_y to the train_y list
#train_y.extend(train_y3) Add the third train_y to the train_y list

tf.reset_default_graph()
net = tflearn.input_data(shape=[None, len(train_x[0])])
net = tflearn.fully_connected(net, 8)
net = tflearn.fully_connected(net, 8)
net = tflearn.fully_connected(net, len(train_y[0]), activation='softmax')
net = tflearn.regression(net)

model = tflearn.DNN(net, tensorboard_dir='tflearn_logs')
model.fit(train_x, train_y, n_epoch=250, batch_size=200, show_metric=True) #It would
take too long to do 1000 epochs of so many intents so the number had to be lowered. The
batch size was increased for more accuracy and lower run times for training
model.save('model.tflearn')
print("model created successfully")
```

An example of how the training step looked like (same format but different values):

```
Training Step: 12400 | total loss: 0.90650 | time: 0.117s
| Adam | epoch: 496 | loss: 0.90650 - acc: 0.7001 -- iter: 781/781
--
Training Step: 12425 | total loss: 0.91117 | time: 0.126s
| Adam | epoch: 497 | loss: 0.91117 - acc: 0.7078 -- iter: 781/781
--
Training Step: 12450 | total loss: 0.93227 | time: 0.127s
| Adam | epoch: 498 | loss: 0.93227 - acc: 0.6662 -- iter: 781/781
--
Training Step: 12475 | total loss: 1.29017 | time: 0.112s
| Adam | epoch: 499 | loss: 1.29017 - acc: 0.6321 -- iter: 781/781
--
Training Step: 12500 | total loss: 0.93543 | time: 0.098s
| Adam | epoch: 500 | loss: 0.93543 - acc: 0.6890 -- iter: 781/781
--
('Final Accuracy:', [0.54871795758222919])
```

For example, my model used 50,000 training steps, had 250 epochs and had 48,000 iterations.

Use of machine learning for responses:

Code using the neural network to generate responses:

```
def response(sentence): #Function which takes in a user input
    results = model.predict([bagOfWords(sentence, words)]) #The sentence is converted
    into a bag of words. This bag of words is then inserted into the model to output a
    set of predictions for the most probable class
    results_index = np.argmax(results) #The index of the highest predicted class is found
    tag = classes[results_index] #The class/tag is found using the index

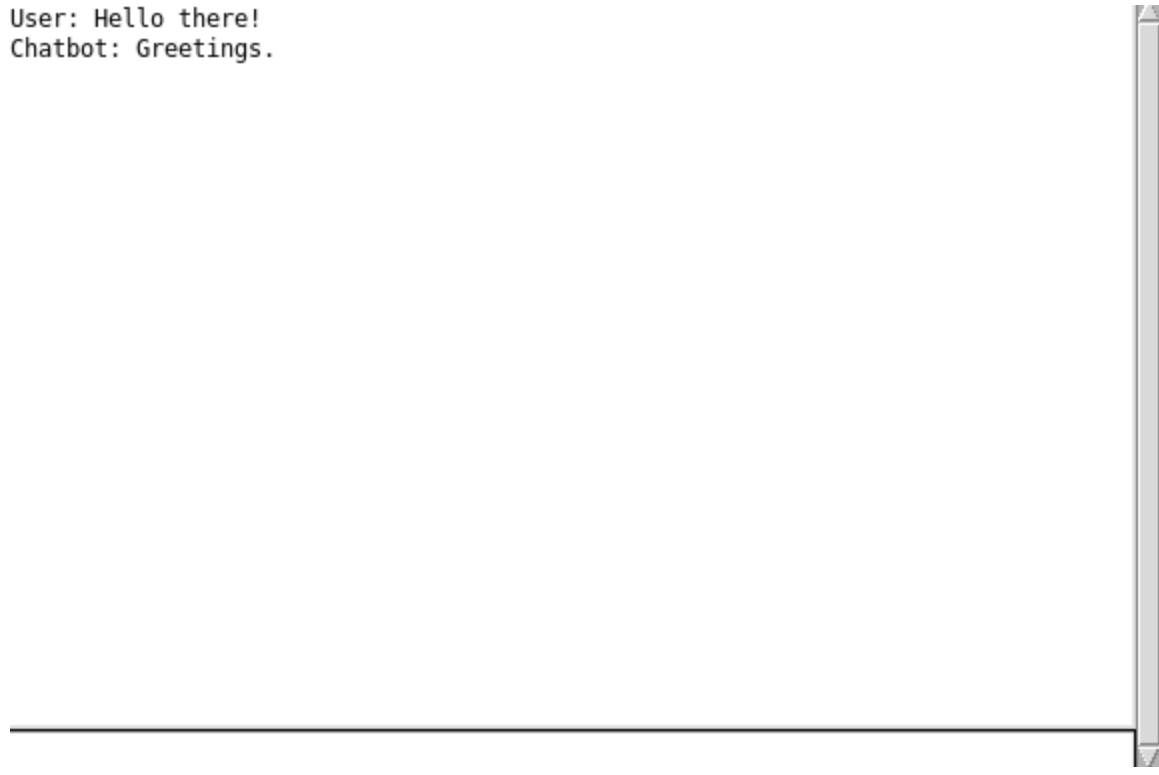
    for intent in intents['intents']: #For each intent
        if intent['tag'] == tag: #If the tag is equivalent to the most probable tag
            response = intent['child'] #Then its child is the response
            break #Breaking from the for loop because the tag was already found so I no
            longer have to iterate through it
    return response #Return the response found
```

Usability Features

The sections in the user interface have been made big so they can be easily seen. They are also separated so they can be differentiated.

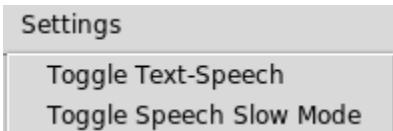
User: Hello there!

Chatbot: Greetings.



The conversation box is very big so it can contain a large amount of text for the conversations. There's a scroll bar so that previous messages can be looked at. There is a small bar which separates the conversation box with the user input box so the user can tell apart the different parts of the user interface. The text has been made to be easily readable - normal black font on a white background with a medium size (to fit more messages in the box without having to scroll).

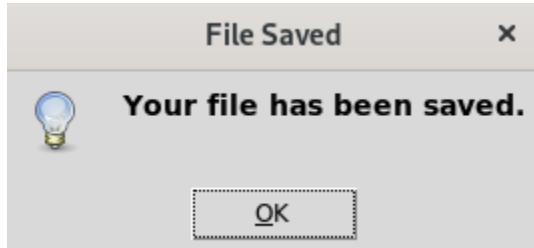
The user interface contains a built-in text-to-speech functionality that can be toggled by the user. This is especially important for users who would normally use screen readers to enable them to read information on a computer. The chatbot's response is read out to them so they can hear what is said. For the case that the audio in the text-to-speech is too fast to be understood, the user can enable slow mode, decreasing the reading rate.



I have added some extra features to help the users which are information dialogs. In the help menu, there are submenus that when clicked, display information in aims of helping the user. For example, if the user is confused on how to start talking to the chatbot (for example, they can't differentiate the two boxes on the user interface so they don't know where to start typing), they could click on the "How to start?" submenu which would open a helpful dialog. The other submenu helps the user if they are stuck on what to tell the chatbot by suggesting possible messages to send.



A final helpful feature is the dialogs for errors and processes. When the user runs something on the menu bar for example to create a new conversation (by refreshing the current one), enabling text-to-speech or others, they will receive a dialog box stating what has been done. If the program runs into an error, the user will see an error dialog box and the program will remain fully functional.



Survey

Person 1:

Name: Emily

How long on average did it take for the chatbot to respond to you?: Too fast to count, it responded immediately before I even had enough time to start a stopwatch.

How long (in messages) was your conversation with the chatbot?: 10 messages from me, 20 from both.

How many messages would you say were accurate to your messages?: 9/10 of the bot's messages

How would you rate the simplicity of using the program?: It was simple to use. There weren't a lot of features in the program so I didn't find it hard to locate where everything was and what to do. It did take a minute or 2 to get used to the user interface but after that, it was really easy to navigate.

Conversation log:

User: Hello there!

Bot: ... Hi.

User: How are you?

Bot: Head still secure to the neck.

User: I guess that is good then.

Bot: Do you know who I am? This is my daughter. I am Frau Weber. We are favoured guests!

User: Favoured guests of who, Frau Weber?

Bot: Now? I don't know. I know that I'll never have the strength to leave you again.

User: That is so cute, I love you too

Bot: Oh.

User: Why do you sound so surprised?

Bot: You shouldn't. You shouldn't trust anyone... certainly not the Attorney General who wants it all for himself. Not the young federal agent who wants your job. Not even the old man who just wants to hang onto what he's created. Don't trust anyone. Just find the Minority Report.

User: Who broke your trust to the point where you don't trust anyone?

Bot: You know what I'm talkin' about. All that mess with ya teachers and... boys in the backs of trucks.

User: Oh wow the teachers and truck boys must have done a hit on you.

Bot: Mr. Chairman, there is an old saying: The enemy of my enemy is my friend.

User: That is a really good saying but unfortunately my name is not Chairman nor am I a mister.

Bot: Yes, but important people are concerned about this. Judge Hillier spoke to Alma Styles --

User: can you remind me who Judge Hillier is again? Oh and Alma Styles too.

Bot: You really think we were set up?

What computer and operating system did you use?: Windows 7 Laptop

Feedback: It took the program around 3 minutes to start up but after it was ready, it worked well with no problems. The chatbot was able to respond really quickly to my messages. The chatbot mostly responds accurately to my questions. It also managed to bring up other topics which directed the conversation so I found that useful. The last message of the conversation, however, wasn't really related to my own message. As well as this, the chatbot used a variety of words and sentence structures so I was exposed to a variety of English techniques.

Person 2:

Name: Mia

How long on average did it take for the chatbot to respond to you?: As soon as I clicked enter after typing in a message, it sent a response.

How long (in messages) was your conversation with the chatbot?: 5 messages from me

How many messages would you say were accurate to your messages?: 4/5 responses

How would you rate the simplicity of using the program?: It was pretty simple to use. The user interface doesn't have a lot to it, neither does it have any difficult aspect. The input box was harder to find than the other features but it only took me like 30 seconds so it's not a big deal. The design of the user interface is simple.

Conversation log:

User: Hello!

Bot: Do you know what I'm doing right now, Peter? I'm looking at the name of the girl I'm going to kill tonight.

User: Woah, don't go killing girls okay?

Bot: I'm impressed. He knows a lot.

User: Why thank you. Though I am a she :)

Bot: I'd better not. It's a bit of a secret.

User: Just don't plot my death just because I am a girl okay?

Bot: Well, sir, if you should do it again, which I hope you won't, please remember the order. Counterpane, blanket, blanket, sheet, sheet.

User: Looks like I have to be a boy because you will murder me. What else would all these blankets and sheets be for?

Bot: They were in a quandary, and--

What computer and operating system did you use?: Windows 10 Desktop Computer

Feedback: Right at the beginning of my conversation with the bot, the bot straight on started talking about wanting to kill people which is pretty random to what I said however, it dragged the conversation along. I could talk to the bot about its plans for killing people instead of going on about "how are you?" and it responds with "good" or something similar, leading to a pretty dull conversation. Because of this, I like the fact that the chatbot doesn't always respond completely accurately, it should respond mostly accurately. I also found it funny when the bot wanted to call me a he to keep my gender a secret as well as having a murderous personality in our conversation. Finally, I did actually find the bot useful for spelling because even in the short period of time that our conversation lasted, words like "impressed" are words that I am not good at remembering how to spell and it was actually used by the chatbot.

Person 3:

Name: Amber

How long on average did it take for the chatbot to respond to you?: Immediately after I sent my messages

How long (in messages) was your conversation with the chatbot?: 15 messages

How many messages would you say were accurate to your messages?: 11/15 messages

How would you rate the simplicity of using the program?: It was pretty easy to use. I just started the program and learnt how to use the interface (which was done during a short period of time). Once I was ready, I dove straight into chatting up with the bot.

Conversation log:

User: Hey, what's up?

Bot: Gentle stuff like talking, sharing thoughts and ideas, secrets and past experiences. Stuff like that, you know.

User: Wow that's so deep and wise.

Bot: Yes, I know.

User: So, do you have anything else that's deep and wise to say to me?

Bot: No, of course I do... It's just that right now I need to --

User: Leave? Are you leaving me? How rude.

Bot: No, sir. I was empathizing. Sorry.

User: No worries, I forgive you.

Bot: George...

User: Who told you my name was George?

Bot: A doll?

User: A doll... told you my name was George?

Bot: I shot at one of them.

User: Ah so now I understand. The doll feared for its life after you shot its friend so it told you that my name is George.

Bot: You're my what?

User: Nevermind, don't worry about it. I already figured it out.

Bot: Inflation, tuition, college. Children are damned expensive. I know.

User: But surely it must be worth it, having a child?

Bot: You've been watching too much TV.

User: I sure do like my TV time. Especially when watching Netflix.

Bot: Well, if it's fun for you, it's all right. I get a kick out of my work when the customers aren't too crabby.

User: If you don't mind me asking, what kind of work do you do?

Bot: What?...Maybe...no...I dunno...

User: Well that is not suspicious at all :)

Bot: No.

User: Only if you say so.

Bot: No. Her mother's awful worried about her. When she left the house she said she was going on a roaring drunk. Er, the girl, I mean!

User: Remind me, when did this happen again?

Bot: We were friends. I trusted him. I should have known it was weird. But, then again I was weird.

What computer and operating system did you use?: Windows 10 Desktop Computer

Feedback: The chatbot was mostly accurate. Whenever it wasn't, it did help me provide prompts to continue the conversation. We talked about several topics in our relatively short conversation that were interesting and it accurately responded to most of my messages.

Person 4:

Name: Emek

How long on average did it take for the chatbot to respond to you?: No time, it responded immediately.

How long (in messages) was your conversation with the chatbot?: I wrote 8 messages

How many messages would you say were accurate to your messages?: 7/8 were accurate

How would you rate the simplicity of using the program?: The program is pretty simple, there's not much to it other than chatting with the chatbot. It was fun playing around with the text-speech and slow mode at times and they were the most complicated parts of the program anyway for me.

Conversation log:

User: Heyo!

Bot: *Oh, I thought there was something funny... You see, Bruce, you don't mind if I call you Bruce, do you? After all, we're practically related --*

User: *I am related to a chatbot? Great... Also, I don't mind being called after Bruce Wayne :3*

Bot: *'At's-a no good. I went to Niagara Falls once.*

User: *How was Niagara Falls (I have never been).*

Bot: *Actually, it's a big secret. Oh, this is delicious! What is it?*

User: *Oh I didn't realise that, what are you eating?*

Bot: *Don't worry, Killer. You'll be all right. I seen enough to know.*

User: *Interesting... so who have I apparently killed?*

Bot: *Jack Manfred.*

User: *How did I kill Jack Manfred?*

Bot: *Right.*

User: *If you don't want to tell me, that's fine. I won't force you.*

Bot: *I see. What do you think our alternatives are?*

User: *I dunno but I do need to go now, goodbye!*

Bot: *Why's it gotta be Tommy!*

What computer and operating system did you use?: Windows 7 Desktop Computer

Feedback: I was very entertained with my conversation with the chatbot, I found it funny that the chatbot was convinced that I murdered “Jack Manfred”. Furthermore, I found this chatbot useful for finding out how to spell words like “Niagara Falls”, it’s commonly pronounced as “Niagra Falls” and that’s how I have been spelling the place therefore the chatbot has helped correct my spelling. Because of this, I think the bot succeeds with its job of teaching a person how to spell words in English. The chatbot was mostly accurate during our short conversation.

Analysis:

On average, the chatbot had 82.7% accuracy during each conversation. This suggests that the chatbot is sufficient enough to respond accurately to the user as most of its messages are considered to be accurate. This is a success for the requirement for the chatbot to reply with a context-related message as the results from this survey implies that the bot is able to respond accurately 70%+ of the time which is a good result for a chatbot. The responders have provided feedback that the chatbot was useful for them, especially for the education aspect which is the main focus on the project. They also commented on the simplicity, not being too hard to understand what’s going on because the concept is relatively clear. This supports the idea that the program is simple to use by the end-user. Several responders have also commented that the chatbot is useful for entertainment purposes, which is another possible use for the chatbot that was discussed earlier in the project when deciding on what this could be used for. According to this survey, the chatbot project has successfully passed the requirements in the success criteria.

Limitations

The biggest limitation is creating the training data. Although this process works well, it takes a long time (2-3 hours) to create the intents dictionary due to there being so much conversation data to run through. However, with knowledge of over 138,000 paired messages, the chatbot has the ability to create suitable and specific responses to the user so even though creating the training data took a long time, this process only occurred once and the huge amount of training data available improves the ability of the chatbot to talk properly. This process was also done before knowing that over 138,000 sets of messages was too much data to train the chatbot with. This process took a long time to occur as the bag of words array was over 100,000 items long to represent each unique word used. This same length of array was used on all of the paired messages which meant that a large amount of data had to be converted into numerical data (very long bag of words) so that it would be suitable to be used by a neural network. This therefore took a long time to process. There is no way to prevent this limitation without decreasing the amount of data used.

Another limitation is that the chatbot doesn't always respond correctly to context. For example, when I asked the bot for its favourite colour, it responded with a completely random comment. However, through my series of tests, I have calculated that the chatbot responds accurately around 70% (most likely between 70-80% according to conducted research in the previous section) of the time, which is a fairly good rating, especially because chatbots are generally not completely accurate. A way to improve this accuracy is to teach the chatbot more intents. This couldn't have been done with the pickling methods used as 48,000 was the maximum that could be pickled then later opened in the training module. I have suggested during the development stage of the response creation and the conversion to training data to use SQL instead of pickling. The only problem with this is that it's very likely that there will be a memory error even with an SQL database because a large amount of data will still need to be loaded in and extracted. As well as this, loading in a very large SQL database may cause memory errors itself. Furthermore, the training times for the chatbot model will significantly increase which won't affect the user but it will make the computer unable to be used until this process has finished (as 80%+ of the memory is dedicated to Python). This could be something to play around with if I had more time to work on the development side of the project. Another possible solution is inserting more RAM into the computer being used so that more than ~16GB of RAM can be dedicated to Python during the training process so more intents can be trained (although, this is not an ideal solution).

Furthermore, an original plan for this solution consisted of the chatbot being able to spell check messages sent by the user, however this was later removed. Having the chatbot being able to spell check meant that the solution could focus further on its impact made on the user for education. This would be because if the user misspells a word which they didn't know how to spell, the chatbot would correct them and they could learn from their mistake. But, there are a lot of limitations when using this: the chatbot will respond with an automated correction message rather than responding to the context, the chatbot could make mistakes by not flagging incorrect words or flagging correct words by accident and the spell checking may not

be accurate enough to correct the word misspelled. Spell checking has also gone against the stakeholders' request of not having any automated messages, they wanted context-related responses. For these reasons, spell checking has been removed from the solutions.

Moreover, to run the main program is a slow process. Answers are not immediate. It could take over 10 minutes for the bot to respond to messages which is not ideal.

A final limitation is that the user has to have some background of the English language to use it. This is because the chat bot will not be able to identify what the user is saying and therefore will not be able to provide good contextual responses. There is no way for me to fix this limitation, it depends on the user who is using this chatbot.

Maintenance

The largest limitation of the chatbot is its inability to always correctly respond to the user. It has a very high accuracy of contextual responses but it could be made higher. This generally shouldn't cause any issues because any user using a chatbot should be aware that a chatbot often makes mistakes, it can't always respond super accurately. Although, what could cause issues is if in a session, the chatbot confuses the user by using several weird statements as responses. This was occasionally seen done in the conversation logs by the survey responders. In order to make the chatbot as context-accurate as possible, I have taught it with as many intents as I possibly could, but in the future, the model could be adapted to handle and be taught more intents.

Further maintenance on improving the chatbot is finding a better way to handle large amounts of data. This could be done by using a SQL database, creating an algorithm for more efficient storage or even making the code more efficient to allow the same process to train even more intents. Once that's done, there may be ways to improve the training process by making it faster for example creating a Tensorflow pipeline and feeding data through it or exploring possible settings for models to improve this process. There are many different types of models that can be created using Tensorflow and TFlearn so perhaps using a different type of model would improve its context predictions. The current model could also be edited to e.g. prioritise certain words over others when making predictions for example stop words (the, a, in, etc.) would not weigh as much as other words (green, ball, hat, etc.) which would provide a more accurate response.

Moreover, in my first survey (to test whether the user interface is considered simple or not), there was feedback received which gave good suggestions to improve the software in a future update. A small documentation was suggested however the chatbot does not have a lot of features therefore it was not included in the first version of the program. Though with additions to the functionality of the software, this could be added to future versions. Several people have suggested making the input field more obvious for example by: adding a placeholder like "type here...", outline the box with a different colour, increase its size or prompt the help section at the start. This is a usability feature that could be added to improve the software and make it easier for new users to get started.

If the stakeholder wishes to add any new features to the program or make any changes, that can be done easily as the program has been split into different modules. These models are separate and can be changed without affecting the other for example a different "response" function would not affect any of the user interface's functionality. Any changes made in different modules will not affect the other (as long as these changes are made after the neural network has been created and changed. For example, if you wanted to add an extra intent to be recognised by the model, the entire training process would have to occur again in order for it to take effect).

The code in each module is split up to make it easy to follow through, common functions and variables are kept close together so changing e.g. certain aspects on the exit button in the user interface can be easily done. This makes it easier for a new developer to edit features on this program.

Future versions of this software could incorporate new features such as a more specific settings menu where the user could decide on the specific number to change the words per minute rate to. It may also provide better support for checking if a loaded conversation file is a supported file for this program and not any random text file. A feature to remove the simplicity of the program (in the user's eyes) would be to add an animated avatar for example to show the chatbot "speaking out" the sentence said. This could add interest to the user to continue learning using the chatbot. Further tools could be added to assist learning for example a link between the program and a dictionary - if the user is confused on what a word that the chatbot has typed means, they could click on a button where the program would search for the definition of the word to make it easier to learn what the word means.

Bibliography

- <https://www.youtube.com/channel/UCfzICWGWWYyIQ0aLC5w48gBQ>. 2017. *Creating A Chatbot With Deep Learning, Python, And Tensorflow Part 1.* [online] Available at:
<https://www.youtube.com/watch?v=dvOnYLDg8_Y> [Accessed June 2019].
- <https://www.youtube.com/channel/UCfzICWGWWYyIQ0aLC5w48gBQ>. 2017. *Creating A Chatbot With Deep Learning, Python, And Tensorflow Part 2.* [online] Available at:
<<https://www.youtube.com/watch?v=K2hFNFN9IRc>> [Accessed June 2019].
- <https://www.youtube.com/channel/UCfzICWGWWYyIQ0aLC5w48gBQ>. 2017. *Creating A Chatbot With Deep Learning, Python, And Tensorflow Part 3.* [online] Available at:
<<https://www.youtube.com/watch?v=gQjvotqlvjs>> [Accessed June 2019].
- <https://www.youtube.com/channel/UCfzICWGWWYyIQ0aLC5w48gBQ>. 2017. *Creating A Chatbot With Deep Learning, Python, And Tensorflow Part 4.* [online] Available at:
<<https://www.youtube.com/watch?v=krxuhSARhAs>> [Accessed June 2019].
- <https://www.youtube.com/channel/UCfzICWGWWYyIQ0aLC5w48gBQ>. 2017. *Creating A Chatbot With Deep Learning, Python, And Tensorflow Part 5.* [online] Available at:
<<https://www.youtube.com/watch?v=9os9uQJqDbI>> [Accessed June 2019].
- <https://www.youtube.com/channel/UCfzICWGWWYyIQ0aLC5w48gBQ>. 2017. *Creating A Chatbot With Deep Learning, Python, And Tensorflow Part 6.* [online] Available at:
<<https://www.youtube.com/watch?v=tw8G8-KMRf8>> [Accessed June 2019].
- <https://www.youtube.com/channel/UCfzICWGWWYyIQ0aLC5w48gBQ>. 2017. *Creating A Chatbot With Deep Learning, Python, And Tensorflow Part 7.* [online] Available at:
<<https://www.youtube.com/watch?v=040CfFRJ9Rs>> [Accessed June 2019].
- <https://www.youtube.com/channel/UCfzICWGWWYyIQ0aLC5w48gBQ>. 2017. *Creating A Chatbot With Deep Learning, Python, And Tensorflow Part 8.* [online] Available at:
<<https://www.youtube.com/watch?v=gFxiQXnt9w4>> [Accessed June 2019].
- <https://www.youtube.com/channel/UCfzICWGWWYyIQ0aLC5w48gBQ>. 2017. *Creating A Chatbot With Deep Learning, Python, And Tensorflow Part 9.* [online] Available at:
<<https://www.youtube.com/watch?v=tw8G8-KMRf8>> [Accessed June 2019].
- Xu, J., 2020. *How To Create A Chatbot With Python & Deep Learning In Less Than An Hour.* [online] Medium. Available at:
<<https://towardsdatascience.com/how-to-create-a-chatbot-with-python-deep-learning-in-less-than-an-hour-56a063bd44>> [Accessed June 2019].
- Fabisak, R., 2019. *How To Create An Intelligent Chatbot In Python.* [online] Medium. Available at:
<<https://chatbotslife.com/how-to-create-an-intelligent-chatbot-in-python-c655eb39d6b1>> [Accessed June 2019].
- Waseem, M., 2020. *How To Make A Chatbot In Python | Python Chatterbot Tutorial | Edureka.* [online] Edureka. Available at: <<https://www.edureka.co/blog/how-to-make-a-chatbot-in-python/>> [Accessed June 2019].

Pandey, P., 2018. *Building A Simple Chatbot From Scratch In Python (Using NLTK)*. [online] Medium. Available at: <<https://medium.com/analytics-vidhya/building-a-simple-chatbot-in-python-using-nltk-7c8c8215ac6e>> [Accessed June 2019].

TensorFlow. 2020. *Guide | Tensorflow Core*. [online] Available at: <<https://www.tensorflow.org/guide>> [Accessed August 2019].

Damien, A., 2020. *Tutorials - Tflearn*. [online] Tflearn.org. Available at: <<http://tflearn.org/tutorials/>> [Accessed August 2019].

Final Code

TrainingData.py

```

import json

movie_lines = []
data = {}
data['intents'] = []
tag = 0

with open('movie_lines.txt', 'r', encoding='iso-8859-1') as file:
    for r in file.readlines():
        temp_list = r.split(' +++$+++ ')
        movie_lines.append(temp_list)

def fileInsertion(parent, child):
    global tag
    tag += 1
    data['intents'].append({
        'tag': tag,
        'parent': parent,
        'child': child
    })
    #print("data appended", tag)
    #print(data, "\n")

def pairing(row):
    conversation_texts = []
    position = -1
    new_list = row.split(' +++$+++ ')
    if len(new_list) > 4:
        return
    lines = new_list[3]
    if len(lines) == 1:
        return
    else:
        line_ids = lines.split(',')
        for line_id in line_ids:
            position += 1
            formatted_id = line_id.replace("[", "").replace("]", "").replace("'", "")
            .replace(" ", "")
            for line_data in movie_lines:
                if formatted_id == line_data[0]:
                    if position == 0:
                        conversation_texts.append(line_data[4].replace("\n", ""))
                    elif position == len(line_ids)-1:

```

```
child = line_data[4]
parent = conversation_texts[position-1]
fileInsertion(parent, child)
#print("parent:" + parent + "child:" + child)
#print(conversation_texts)
else:
    conversation_texts.append(line_data[4].replace("\n", ""))
    child = line_data[4].replace("\n", "")
    parent = conversation_texts[position-1]
    fileInsertion(parent, child)
#print("parent:" + parent + "child:" + child)
#print(conversation_texts)

with open('movie_conversations.txt', 'r', encoding='iso-8859-1') as f:
    for row in f.readlines():
        pairing(row)

with open('intents.json', 'w') as outfile:
    json.dump(data, outfile)
#print("File creation complete")
```

NumericalData.py

```

from nltk.tokenize import word_tokenize
from nltk.stem.lancaster import LancasterStemmer
import json
import string
import numpy as np
import pickle
punc_dic = dict((ord(punct),None) for punct in string.punctuation)
words = []
classes = []
documents = []

with open('intents2.json') as json_data:
    intents = json.load(json_data)

for intent in intents['intents']:
    pattern = intent['parent']
    w = word_tokenize(pattern)
    words.extend(w)
    documents.append((w, intent['tag']))
    classes.append(intent['tag'])

stemmer = LancasterStemmer()
words = [stemmer.stem(word.lower()) for word in words]
words = [word.translate(punc_dic) for word in words]
words = sorted(list(set(words)))
words.remove('')

#print(len(documents), "documents")
#print(len(classes), "tags", classes)
#print(len(words), "unique stemmed words", words)

training = []
output_empty = [0] * len(classes)

for doc in documents:
    bag = []
    parent_words = doc[0]
    parent_words = [stemmer.stem(word.lower()) for word in parent_words]
    parent_words = [word.translate(punc_dic) for word in parent_words]
    while '' in parent_words:
        parent_words.remove('')
    for w in words:
        bag.append(1) if w in parent_words else bag.append(0)

```

```
output_row = list(output_empty)
output_row[classes.index(doc[1])] = 1

training.append([bag, output_row])

training = np.array(training)
train_x = list(training[:,0])
train_y = list(training[:,1])

pickle.dump( {'words':words, 'classes':classes, 'train_x':train_x, 'train_y':train_y},
open( "training_data", "wb" ) )
#print("data pickled successfully")
```

Training.py

```
import pickle
import tensorflow as tf
import tflearn
training_data1 = pickle.load(open("training_data", "rb"))
train_x = training_data1['train_x']
train_y = training_data1['train_y']

tf.reset_default_graph()
net = tflearn.input_data(shape=[None, len(train_x[0])])
net = tflearn.fully_connected(net, 8)
net = tflearn.fully_connected(net, 8)
net = tflearn.fully_connected(net, len(train_y[0]), activation='softmax')
net = tflearn.regression(net)

model = tflearn.DNN(net, tensorboard_dir='tflearn_logs')
model.fit(train_x, train_y, n_epoch=250, batch_size=200, show_metric=True)
model.save('model.tflearn')

#print("model created successfully")
```

Model.py

```

from nltk.tokenize import word_tokenize
from nltk.stem.lancaster import LancasterStemmer
import numpy as np
import tflearn
import tensorflow as tf
import json
import pickle
import string

training_data = pickle.load(open("training_data", "rb"))
words = training_data['words']
classes = training_data['classes']
train_x = training_data['train_x']
train_y = training_data['train_y']
punc_dic = dict((ord(punct),None) for punct in string.punctuation)
stemmer = LancasterStemmer()

with open('intents2.json') as json_data:
    intents = json.load(json_data)

def cleanUp(sentence):
    sentence_words = word_tokenize(sentence.lower())
    sentence_words = [word.translate(punc_dic) for word in sentence_words]
    while '' in sentence_words:
        sentence_words.remove('')
    sentence_words = [stemmer.stem(word.lower()) for word in sentence_words]
    return sentence_words

def bagOfWords(sentence, words):
    sentence_words = cleanUp(sentence)
    bag = [0] * len(words)
    for s in sentence_words:
        for i,w in enumerate(words):
            if w == s:
                bag[i] = 1
    return(np.array(bag))

tf.reset_default_graph()
net = tflearn.input_data(shape=[None, len(train_x[0])])
net = tflearn.fully_connected(net, 8)
net = tflearn.fully_connected(net, 8)
net = tflearn.fully_connected(net, len(train_y[0]), activation='softmax')
net = tflearn.regression(net)

```

```
model = tflearn.DNN(net, tensorboard_dir='tflearn_logs')
model.load('./model.tflearn')

def response(sentence):
    results = model.predict([bagOfWords(sentence, words)])
    results_index = np.argmax(results)
    tag = classes[results_index]

    for intent in intents['intents']:
        if intent['tag'] == tag:
            response = intent['child']
            break
    return response
```

UserInterface.py

```

import tkinter as tk
from tkinter import messagebox
from tkinter.simpledialog import askstring
from tkinter import filedialog
import importlib
import pyttsx3

class Application(tk.Frame):
    def __init__(self, name):
        tk.Frame.__init__(self)
        self.name = name
        self.name.title("Chat Bot")
        self.option_add('*tearOff', tk.FALSE)
        self.menu = tk.Menu(self.name)
        self.name.config(menu = self.menu)
        self.file = tk.Menu(self.menu)
        self.help = tk.Menu(self.menu)
        self.settings = tk.Menu(self.menu)
        self.menu.add_cascade(label = "File", menu = self.file)
        self.file.add_command(label = "New", command = self.restart)
        self.file.add_command(label = "Load", command = self.load)
        self.file.add_command(label = "Save", command = self.saveText)
        self.file.add_command(label = "Exit", command = self.quit)
        self.menu.add_cascade(label = "Help", menu = self.help)
        self.help.add_command(label = "How to start?", command = self.startBox)
        self.help.add_command(label = "Starting Prompts", command = self.promptBox)
        self.text_to_speech = tk.IntVar()
        self.text_to_speech.set(0)
        self.slow_mode = tk.IntVar()
        self.slow_mode.set(0)
        self.settings_menu = tk.Menu(self.menu)
        self.settings_menu.add_checkbutton(label = "Toggle Text-Speech", onvalue = 1,
                                         offvalue = 0, variable = self.text_to_speech, command = self.toggleSpeech)
        self.settings_menu.add_checkbutton(label="Toggle Speech Slow Mode", onvalue = 1,
                                         offvalue = 0, variable = self.slow_mode, command = self.toggleSlow)
        self.menu.add_cascade(label = "Settings", menu = self.settings_menu)

        self.exit_photo = tk.PhotoImage(file = "leave.png")
        self.leave = tk.Button(self, image = self.exit_photo, command = self.quit)
        self.leave.pack(side = tk.TOP, anchor = tk.W)

        self.img = tk.PhotoImage(file = "avatar.png")
        tk.Label(image = self.img).pack()

```

```
self.engine = pyttsx3.init()
self.bot = importlib.import_module("Model")

self.scroll = tk.Scrollbar(self)
self.scroll.pack(side = tk.RIGHT, fill=tk.Y)
self.messages = tk.Text(self, wrap = tk.NONE, yscrollcommand=self.scroll.set)
self.messages.config(state = tk.DISABLED)
self.input_user = tk.StringVar()
self.input_field = tk.Entry(self, text = self.input_user)
self.input_field.bind("<Return>", self.enterPressed)
self.messages.pack()
self.scroll.config(command = self.messages.yview)
self.input_field.pack(side = tk.BOTTOM, fill = tk.X)

def enterPressed(self, par1):
    self.input_get = self.input_field.get()
    self.messages.config(state = tk.NORMAL)
    self.messages.insert(tk.INSERT, '%s\n' % ("User: " + self.input_get))
    self.input_user.set('')
    self.response = self.bot.response(self.input_get)
    self.messages.insert(tk.INSERT, '%s\n' % ("Chatbot: " + self.response))
    self.messages.config(state = tk.DISABLED)
    if self.text_to_speech == 1:
        if self.slow_mode == 1:
            self.engine.setProperty('rate', 125)
            self.engine.say(self.response)
            self.engine.runAndWait()
        else:
            self.engine.setProperty('rate', 200)
            self.engine.say(self.response)
            self.engine.runAndWait()

    return "break"

def restart(self):
    self.messages.config(state = tk.NORMAL)
    self.messages.delete('1.0', tk.END)
    self.messages.config(state = tk.DISABLED)
```

```

def load(self):
    self.file_name = filedialog.askopenfilename(initialdir = "./", title = "Select
    conversation file to load",
                                                filetypes = (("Text files","*.txt"),
                                                ("all files","*.*")))
    if self.file_name == None or self.file_name == '' or self.file_name == ' ':
        return
    try:
        self.file_name = self.file_name.split('/')
        self.file_length = len(self.file_name)
        self.file_name = self.file_name[self.file_length-1]
        with open(self.file_name, 'r') as file:
            self.conversation = file.readlines()
            self.messages.config(state = tk.NORMAL)
            self.messages.delete('1.0', tk.END)
            for line in self.conversation:
                line = line.strip('\n')
                self.messages.insert(tk.INSERT, '%s\n' % (line))
            self.messages.config(state = tk.DISABLED)
    except:
        messagebox.showerror("File Error", "There was an error gathering your file,
        please try again.")

def saveText(self):
    self.file_name = askstring('Name your file', 'What do you want to call this
    file?')
    if self.file_name == None or self.file_name == '' or self.file_name == ' ':
        return
    try:
        with open(self.file_name+'.txt', 'w') as file:
            self.conversation = self.messages.get("1.0", tk.END)
            file.write(self.conversation)
        messagebox.showinfo("File Saved", "Your file has been saved.")
    except:
        messagebox.showerror("Saving Error", "There was an error with saving your
        file, please try again.")

def startBox(self):
    messagebox.showinfo("How to start?",
                        "To start, simply type in the input box (the smallest
                        box/the bottom box) a message and the bot will respond!")

```

```
def promptBox(self):
    messagebox.showinfo("Starting Prompts", "Suggested prompts:\n 'Hey!'\n 'How are
you?'\n 'Are you real?' \n 'Can I be your friend?'")

def toggleSpeech(self):
    print("Text to speech not added to the project yet :(")
    enabled = self.text_to_speech.get()
    try:
        if enabled == 1:
            messagebox.showinfo("Text-Speech", "Text-Speech has been enabled!")
        elif enabled == 0:
            messagebox.showinfo("Text-Speech", "Text-Speech has been disabled!")
    except:
        messagebox.showerror("Text-Speech Error", "There has been an error. Your
setting did not take affect.")

def toggleSlow(self):
    print("Text to speech not available yet")
    enabled = self.slow_mode.get()
    try:
        if enabled == 1:
            messagebox.showinfo("Slow Mode", "Slow Mode for Text-Speech has been
enabled!")
        elif enabled == 0:
            messagebox.showinfo("Slow Mode", "Slow Mode for Text-Speech has been
disabled!")
    except:
        messagebox.showerror("Slow Mode Error", "There has been an error. Your
setting did not take affect.")

if __name__ == "__main__":
    root = tk.Tk()
    Application(root).pack()
    root.mainloop()
```