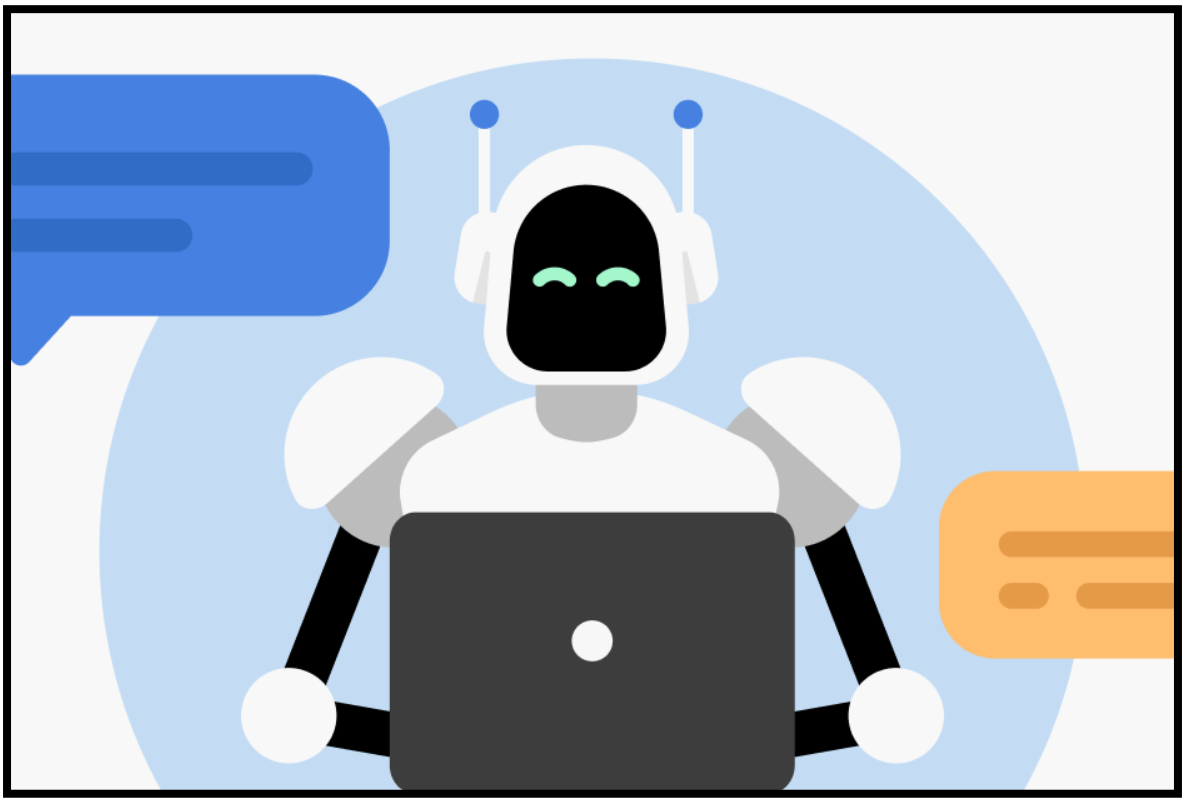
Information technologies

Final examination



Ruben Spiga

June 2024 session

Resources

https://jsons.readthedocs.io/
https://docs.python.org/3/library/difflib.html
https://www.w3schools.com/howto/howto_js_popup_chat.asp
https://www.w3schools.com/js/js_switch.asp
https://www.w3schools.com/html/default.asp
https://www.w3schools.com/css/default.asp
https://www.w3schools.com/js/default.asp
https://www.w3schools.com/js/js_whereto.asp
https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/addEventListener
https://cloud.google.com/vertex-ai/generative-ai/docs/model-reference/gemini https://www.geeksforgeeks.org/how-to-change-the-id-of-element-using-javascript/amp/
https://www.w3schools.com/howto/howto_js_toggle_hide_show.asp

The concept behind the product:

All projects have a readme.txt file to explain their operation. I will attach them at the end of this document.
Throughout this code, I refer to project as either subject/model A/B/C
The only differentiator is the Letter.
Pages 10-11 are copies of the readme.txt files in each product file; please disregard if you have already read them.

For this assignment, I have chosen to build three chat-bot products, all with the same purpose, but using different technologies. Model A uses programmed questions in order to learn questions and responses, this type of bot is better to handle customer services for businesses that get asked specific questions, such as order tracking, as it is easy to link this type of bot to external parcel tracking API's. Model B is built using the Google gemini 1.5-pro API to ensure that it is able to answer a wide variety of questions. This type of bot is better for businesses who want to be able to answer a wide variety of customer needs, and not just to be used for data related requests and Model C is a JS based bot that uses preprogrammed questions that take user input as a series of buttons by using switch statements.

The idea is that these chatbots can be deployed to the website of businesses to handle customer service enquiries with varying levels of complexity, depending on the model selected.

I wanted to make this product as I understand in todays hyper-digitalised world, technology can be used to automate and refine tedious tasks, such as handling basic CS questions, and so, this product is not only relevant to todays world, but is a product with actual value and utility.

Who this product is intended for:

As chat bot programmes, both of my products are intended to be used by any business who has an online presence and would like to automate their basic customer service tasks.

The main deciding factor between the two models for the business is budget. Model A runs without operating costs, aside from server costs. This is due to the fact that it learns all of its behaviours from user programming, whereas model B relies on an external API, namely, Google Gemini Pro 1-5, to facilitate its requests, this means that each request has a cost to the company. For small businesses, I would recommend them to use Model A, whereas for business with larger operations(and budgets), they may be better using Model B as it is able to answer unexpected questions, whilst model A can only answer what it has been programmed to respond to (or similar requests if it recognises them as such). Model C combines the best of both worlds and limits the users selection to what is programmed into the buttons; it is also built with .js which means that the code is easy to deploy to a site, whereas it is a bit more difficult to use flask to deploy .py projects to a site.

Constraints model A:

Model A, thanks to its simplicity, is very easy for any business to implement. It can function in two ways, either learning responses via conversational programming, or by having questions and answers written directly into the knowledge base. This ease of setup makes it so that non-developers are also able to maintain and update the bot, without the need for a developer to come in and do It for them.

Pros of this model:

- It can answer specific questions about the business
- It can be programmed with specific responses
- It can be easily linked to external apis for further utility
- It has open input so the user may ask whatever they like*

Cons of this model:

- If It doesn't recognise a question; it cannot answer
- If a question is written in strange way, the bot may not recognise it
- It has open input so the user may not know what to ask meaning there is a chance the bot will not understand*
- Py is a bit more complex to deploy to websites using flask.

Constraints model B:

I used the prompting function to send a command to the api to only answer questions about "its role as a customer service agent" and so, if you try to ask it questions that are not related to customer service; the bot will fail to answer and will remind the user that as customer service agent, it will only answer work-related questions. This is partially implemented in order to stop people from using the bot for questions unrelated to the business, as you can with AI models, and so, preventing unnecessary requests that ultimately, cost the business money

Pros of this model:

- Due to it being AI powered, it can answer a large number of questions
- The bot responds in a friendly way that adapts to the question asked thanks to the AI
- It can be easily linked to external apis for further utility
- It has open input so the user may ask whatever they like*

Cons of this model:

- If the client doesn't programme in/prompt the questions properly, the bot will not be able to answer the questions accurately. The client must edit the prompt that is send to the API in order for it to have the correct data; this can be done by sending all relevant details to the

bot, or by telling to to limit itself to a set number of questions that you will provide the answers for. You can also have it analyse your site to find info.
- If a question is written in strange way, the bot may not recognise it.
- It has open input so the user may not know what to ask meaning there is a chance the bot will not understand*.
- Py is a bit more complex to deploy to websites using flask

Constraints model C:

Model C is very similar to Model A in that it relies on pre-determined questions and answers to output a response, this is both a benefit and a drawback, depending on how you choose you view it. Model C also uses a restricted input system of buttons which simplifies the user experience, which again, has its pros and cons.

Model C started as an attempt to re-code model A in Java for easier web deployment.

Pros of this model:
- It can answer specific questions about the business
- It can be programmed with specific responses
- It has restricted input to benefit UX with simplicity
- It prompts the user to call a helpline if their issue can't be fixed by the bot.
- It is made in .js so it is easy to deploy to websites.

Cons of this model:
- If the bot doesn't have the correct category or question, it can't assist the user.
- It is a more complex and longer piece of code that relies on .html, .css, and .js to be able to function correctly; for example, this makes it harder to update as if you want to change a buttons content and styling, you have to do it in all 3 languages.

What this product does:

In effect, all three models that I made for this project achieve one goal; to provide automated customer support. There are three models that all take a different route, to achieve more or less the same outcome. All three models are able to take common user FAQ's and answer them based off of programmed knowledge, or knowledge gathered from an API

How it was developed:

Subject A)
When initially building subject A) of the product, I started by making some research about the product I wanted to create and I realised that I could either use python, or javascript to create this project, I opted to use python as at the time, I had a greater understanding of it.

The bot itself comprises of two main sections, the main.py file, and the knowledge.json file. The main.py file is where the actual code for the bot is stored, this is the file that is ran. Secondly, we have the knowledge base, this file is used to store pre-programmed, and learnt queries/responses for the bot.

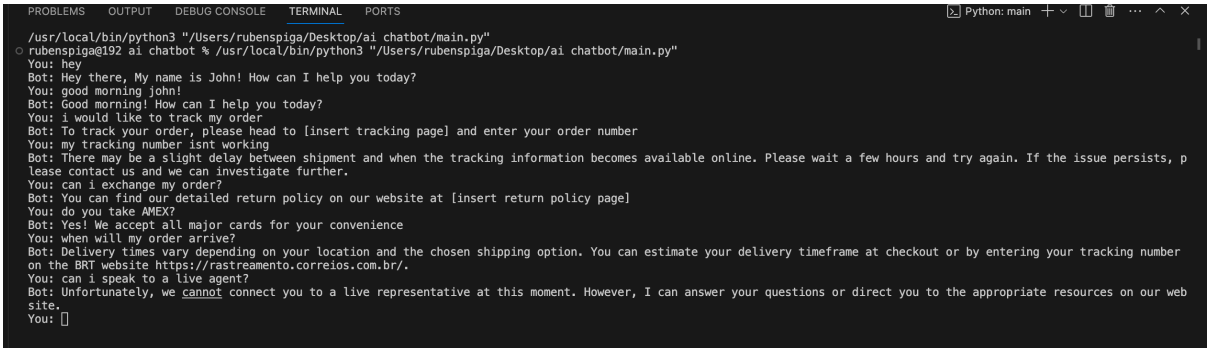
The next stage of the development was the specific research stage, in which I researched different methods of coding and the use of varying python libraries.

I then began to build the project using the “difflib” standard python library in order to use the get_close_matches function to be able to ensure that my bot is able to answer questions that it doesn’t have directly programmed into it, but rather, bares similarity to another learnt question and response, for example, if I programme the question “Where is my order?” with the response “To track your order, please head to [insert tracking page] and enter your order number”, it can then recognise questions that are similar and give the same response. The level of creativity/specificity in these matches comes from the defined “cutoff=0.6” at the end of line 17. The cutoff argument in the get_close_matches function essentially is used to control the strictness of finding similar matches. It can be set at a value between 0 and 1, and in the case of this code, 0.6. This value is known as a good default value as it serves as a mid point between strictness and leniency, ensuring you get reasonably close matches without introducing too many irrelevant ones.

The next part of building my product use the “def” function to define different elements within the code and what each operant does, this dictates the behaviour of how the bot saves new learnt responses to the knowledge base for future usage. Using the def function is a key part of the basic python code as it ensures that each element is specifically programmed in order to perform a desired function. On line 27, there is a good example of a more complex version of the use of the def function, within this section, I define the individual behaviours of the chatbot itself using the if, else, break, and while functions. The use of these functions allows the bot to have several different behaviours that can be executed, depending on the value of the received input, this prevents the code from giving errors when ran as it eliminates possibility for a response to be entered that the bot is not programmed to handle(which would cause a bug in the code).

Several other key functions are also used in this code, such as ‘print’, which is used to display the bots responses, without this function, the bot would have a response to give, but the user would never receive it.
The bot is designed to be programmed via a company operator writing questions into the knowledge-base.
If the bot doesn’t know an answer, or the user sends a blank message, the bot will prompt the user to try to ask their question again; this is done to avoid the bot crashing when it receives an input it doesn’t understand.

In the initial stages of my development, I had the project running in the terminal, but I thought, as a customer service bot, it must run on a website, you can see the original version pictured below



As I began to research ways to run this product online, I discovered the ability to use the python flask library in order to compress my code into an easily executable web app. Upon further research; I discovered that it was a more appropriate solution to create a similar chatbot via JavaScript and so, I began to research how to do this. After trying and having issues; I decided to leave this project in terminal and to begin to learn basic js and rebuild It in that architecture.

Subject B:

Subject b is the simplest of the three models due to the fact that it uses gemini api to function and handle the bulk of the question logs, making it far more lightweight and scalable than model a, or c, which require questions to be coded into it; the Gemini api gives you the base code once you have set it up.

The most important function that I implemented the use of the .env function to be able to export and separately file the API key, and subsequently call it back in. The default google boilerplate code does not have this functionality and it is a security risk to have the API key in the file directly. In order to make the project reply appropriately, I added filtering for explicit content amongst others, to ensure all responses are appropriate. To ensure relevancy, I sent command to the google api giving It the task, tone, and scope of questions to make sure it cannot answer irrelevant questions to prevent the waste of requests (which cost money). The command to send the instruction can be seen below.

```
chat = model.start_chat(history=[])
instruction = "In This chat, your name is Simon, you work For the The Demo Shop; speak English and also if asked, Italian. As Simon, you must repo
```

A series of print statements are also used throughout to print the reply of the bot.

- In order to make this model work, I used the following imports and external libraries:
  import os
- import google.generativeai as genai
- from dotenv import load_dotenv

Subject C:

When creating model C I followed the exact same format of using preprogrammed questions, as well as basic logic that links the JS backend of the code, to the front end that I built in HTML and CSS. I use the const questions = [functions to add in my questions in a question :

answer format, commenting each block based on their category. I then use several different functions, such as

1. Toggle chat to program the button that opens the chatbot on the page
2. getAnswer to utilise the user input constant to pull the user response into the chatbot
3. displayMessage to display the user message

And so on. Every section of code within my project is commented to demonstrate an understand of exactly what it does.

During my development of model c, I came into a critical fault, that is that I was not able to get it to recognise close matches like I did in python so I began to think of a solution. In the end, what I did was remove the ability for the user to type directly, and instead, I use a sequence of buttons and switches in order to create a sequence system where the bot presents the user with an initial start chat button, then 5 initial categories, followed by 4 questions within each of those 5 categories, pertaining to the desired input of the user. When a question is selected from the third set of buttons (inside one of the categories) the bot will give the answer, and then follow up asking if your problem has been solved. You can then click yes, no, or ask a new question. Yes ends the script and thanks the user, No ends the script and prompts the user to contact a helpline number, and ask a new question resets the bot back to the initial 5 categories. The code snippet below shows the logic behind the follow up questions.

```
//the response to the yes or no question sent by the user
function followUpResponse(answer) {
  const botSound = document.getElementById('botSound');
  if (answer === 'yes') {
      displayMessage('🤖 Thank you for using our service!', 'bot');
      document.getElementById('followUp').style.display = 'none';
  } else if (answer === 'no') {
      displayMessage('No');
      displayMessage('🤖 Please call our support line on: 32999999');
      document.getElementById('followUp').style.display = 'none';
  } else if (answer === 'Ask a new question') {
      displayMessage('🤖 How can I help you today?', 'bot');
      document.getElementById('followUp').style.display = 'none';
      document.getElementById('initialQuestions').style.display = 'block';
  }
```

Toward the end of my development , I decided that I wanted to tidy an element of my code so I put my questions into their own file and I call them back into the index file by importing questions.js, above script.js, at the very bottom of the body tag; this is said to improve page load times. This ensures that the front end buttons on the index page are still able to communicate and receive data from the questions; ensuring they run smoothly.

I also programmed the user, and bot to have a different pop sound that plays on message send; this is a simple UX feature that ensures the user that their messages are being sent and

that they are receiving replies, these sounds are then coded into message functions, with a different sound being played dependent on who is sending the message.

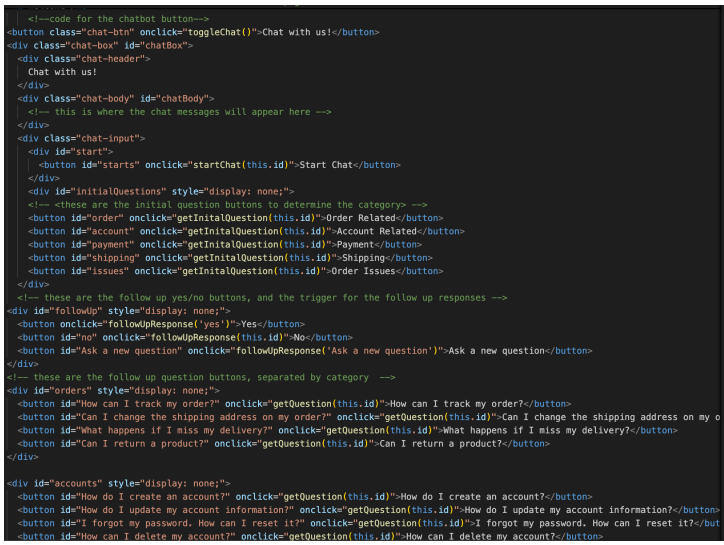
Model C FrontEnd structure:

For the project, the front end is built in HTML, and CSS. The html is used to give basic structure to the page, such as the head, and body. It also sets page scale, title, adds a favicon to the page, and other important things when building a webpage.

```
<!DOCTYPE html> <!-- this top part is the basic setup of the page -->
<html lang="en">
  <head>
    <link rel="stylesheet" href="style.css"> <!-- this links the index to the style sheet-->
    <title>Demo</title> <!-- this sets the tab title -->
    <link rel="icon" type="image/x-icon" href="../images/smilefavicon.png"> <!-- this adds a favicon -->
    <meta charset="utf-8"> <!-- this allows  my code to use the robot emoji , essentially recognising characters that aren't a part of the tradit
    <meta name="viewport" content="width=device-width, initial-scale=1">  <!-- this ensures that the page resized correctly to different devices--
  </head>
```

I used basic html and css structure that I learnt in class; and from w3 schools to build a landing page that I display my live chat within I also chose to use a welcome text that goes clear on hover, and a video background that I use z index to make sure it appears behind other elements **(z index is also used on other css elements to ensure that they appear in-front of the background)**. The most important function on the html are the front end manifestation of the buttons. I have included a snippet below. I included these commands in order to make visual buttons that I then style with css and link to the back-end logic in js. Each button given a button id which defines each specific button by a name, to then be programmed with a function, such as assigning an answer to a question, on the .js backend. I also use button onClick to ensure that the correct function occurs when a button with a specific ID is interacted with.

When clicked, the chat with us button opens the chat box and displays the first visual button, the start button which is used to initiate the chat.
This then goes down the buttons in order ensuring the following flow is followed



Start chat -> Select category -> Select question -> Ask if the users problem was solved
If question is solved, user clicks ye and the bot thanks the user and ends the script
If else question is not solved, and the user clicks no, the bot prompts the user to contact the hotline.
If else the user would like to ask a new question, they click ask a new question and the user is put back to the select category screen

Css is also used to style everything. I went through and individually added colours, margins, padding etc. depending on the needs of each specific element.

Read me. Txt files

Model A:

Hello! This is a brief instruction set about how to use this product!

Product description:

- This project is a virtual customer service assistant that uses a pre-determined set of questions and responses.
- The bot can respond to many customer service related inquiries and it is able to
- recognise non exact matches thanks to the get_close_match library.

Features:

- This project utilises a set of common customer FAQ's and programmed responses
- It can recognise close matches
- If the user sends an empty response; it will ask them to give a new one.

Usage:

1. To activate the code, you must run it in terminal
2. It will then start and "You:" will appear.
3. You will write a response in you, and the bot will then respond

Configuration:

- The questions may be changed by the business by updating the knowledge_base.json file

Dependencies:

- Python (>= 3.6)
- - json: Python library for working with JSON data.
- difflib.get_close_matches: Python library for finding the best matches for a given query within a list of strings.
-

Model B:

Product description:

This project is a virtual customer service assistant that is connected to the Gemini 1.5pro API The assistant is designed to provide responses to customer service related inquiries. the prompt can be changed to include constraints for specific questions, as well as their answers

Features:

- This project utilises the Gemini 1.5-pro api to process your questions and generate responses.
- Implements safety settings to ensure that responses are free from offensive or inappropriate content.
- Allows customisation of response generation parameters such as temperature, top_p, and top_k.
- Provides a user-friendly interface for interacting with the virtual assistant via command-line input.

Usage:

1. Install and unpack the zip file ensuring all dependencies are present. Make sure the API key is in the .env file
2. Run the main.py script to start the virtual customer service assistant in the terminal.

3. You can now ask Simon questions when prompted and he will reply.

Configuration:

- API_KEY: You must set your Gemini API key in the provided.env file.
- Generation Config: You can customise the generation settings in the generation_config area.
- Safety Settings: Configure safety settings for blocking harmful content in the safety_settings list.

Dependencies:

- google-generativeai: This is the .py library for interfacing with the Gemini API.
- python-dotenv: This is the .py library for managing environment variables(env).

Model C:

Product description:

- This project is a virtual customer service assistant that uses a pre-determined set of questions and responses, interacted by using buttons on-screen.
- The bot can respond to many customer service related inquiries.

Features:

- - This project utilises a set of common customer FAQ's and programmed responses.
- - It is integrated into a HTML/CSS landing page for easy access.
- - The front end provides a user-friendly interface for interacting with the virtual assistant.

Usage:

1. To activate the code, please navigate to index.html and click the "Go Live" button in the bottom right corner of your IDE (VScode).
2. You can then click "Chat With Us" to open the chat window, you then have to click "Start chat" and then carry on clicking the relevant buttons based off of your desired selection.
3. When you are asked a question, the bot will follow up and ask if your question is solved
4. Clicking "yes" will end the script and give a thank you message
5. Clicking "no" will end the script and prompt you to call a fake helpline number for more support
6. Clicking "Ask a new question" will restart the script and bring you back to the initial help category selection button selection

Configuration:

- The questions may be changed by the business, as well as then updating the buttons to correspond

Dependencies:

- You must ensure that you are running node js version v20.13.1
- Preferably use the bot with Google Chrome Version 125.0.6422.142 (Official Build) (arm64) or higher