Chatbot Project Report – COSC 310 Individual Project

Individual Developer: Gabriel McLachlan
86257383

Original Team: Team 31

Mohammed Al-Surkhi
Jordan Colledge
Gabriel McLachlan
Jordan Ribbink
Nathan Wright

Project Description:

This project consists of a chatbot built with Electron, comes with a generic mobile interface for simplicity of use and understanding. Primarily utilizing Python and Typescript, this chatbot takes on the role of a medical doctor/remote first responder, whereas the user takes the role of a patient.

The user can ask questions, list symptoms, and in response, the bot will describe the likely illness and potential remedies.

Withing this fork of the project, the chatbot utilizes the power of the internet and various APIs to help the user search up terms and concepts where they may be confused, and suggest hospital inquiry.

My GitHub Username: gmclachlan45

Repository URL: https://github.com/GMcLachlan45/chatbot-app

Documentation for API's used

Wikipedia API

The Wikipedia API has been integrated into the chatbot so that if the user wants to know a bit more about what the Doctor is talking about, all they need to put is "Look up _____", and if it's medical related, the Bot will pull up the info from Wikipedia.

The lookup simply uses the function to get the page's information but determining whether something is related to medicine is actually much more subtle. It looks through the categories of the topic and sees if there are specific keywords within the categories.

This helps to fill out where the team identified were some big gaps in conversation from A2 and A3. If the Bot doesn't know something, then it can search it up online. The ability to distinguish between medical topics also adds to the more human side of the Bot.

The Google APIs

All 4 of the next APIs used were integrated all together to give directions to a dynamically chosen location (the closest hospital to the user). If the user is feeling particularly bad, or if the bot detects that something sounds particularly bad, then the user can inquire about the directions to the nearest hospital.

This works to add an extra, more personal and interactive feature to the Bot, one that grounds it more in the real world then a set of relatively disconnected back and forths.

Though each individual api isn't extensively used, they all have their queries, parsing and formatting. More information below.

Google Geolocation API

Geolocation is a tool that uses nearby cell towers and wifi modems to triangulate the users latitude and longitude with a fair bit of accuracy (not 100% in rural places, like UBCO).

Within the getDirections() function to find the general latitude and longitude of the user. This is then converted into a string for further use within the function.

Google Geocoding API

Reverse Geocoding is the process of taking a set of coordinates and approximating a formatted address based on nearby roads and other addresses.

Within the system, we take the latitude and longitude found via geolocation, and convert it into a named location. This location is output along with the rest of the directions, and lets the user know that the Bot knows the approximate location, which can help give a sense of security that a scared and sick user may need.

Google Places API

The Places API is a useful tool to find relevant destinations. In general this could be used to determine points of interest based on what the user wants to go to, but for the Bot he is restricted to giving directions to the nearest hospital.

For this system, we take the position of the user, and use it as an origin location to bias the results to the closest and most relevant result available. Using Places itself helps to make the system more dynamic, as depending on where you are, the nearest hospital will change, making the system much more flexible.

Google Directions API

Directions is self explanatory. Once we have the user's location, and the hospital's address, the Directions API determines the fastest path between the two and the directions that need to be taken to get there.

Once the directions to the nearest hospital have been determined, the system takes all of the relevant strings, brings them together, removes the html parts and concatenates it with the rest of the Bot's response.

Though the wall of text isn't ideal, it makes much more sense that someone would be typing this out beforehand rather than in small segments. Even a user that didn't have GPS could take those instructions print them out, and follow them to where they needed to go. I also personally believe that this makes him a lot more like a first responder, and a lot more human.

API Conclusion

Those were the 5 APIs integrated into my final rendition of Doctor Phil.

The Wikipedia integration helps to extend conversational potential, adds to its humanness through asking about relevance, and help to clarify anything that the Doctor has said.

Meanwhile, the Google Maps integration helps to add a flexible, reliable and useful tool to the Doctor's coversational toolset while making him seem more human and part of this world.

Further inquiry can be had through looking at the implementation in src/agent/agent.py.

Installation and Usage

Requirements

- Node JS - https://nodejs.org/en/
- NVM (optional) - https://github.com/nvm-sh/nvm
- Python 3 - https://www.python.org/downloads/
- Pyenv (optional) - https://github.com/pyenv/pyenv
- Pyenv-virtualenv (optional) - https://github.com/pyenv/pyenv-virtualenv
- Google Maps Services API key (optional)
  - https://developers.google.com/maps/documentation/javascript/get-api-key

Open terminal in the root of the project and run this command:

1. Install NPM dependencies

➢ npm install

2. Install Python dependencies
   - OPTIONAL. Install Pyenv & Pyenv-virtualenv to manage Python environment. Follow instructions provided within documentation

➢ Create a new virutalenv in Python 3.8.10 and activate it
➢ pyenv virtualenv 3.8.10 ${YOUR_VIRTUALENV_NAME}
➢ pyenv activate ${YOUR_VIRTUALENV_NAME}
   - Install requirements via pip by running the following command from the root folder of the project

➢ pip install -r requirements.txt

3. Run this command to train the bot's neural network:
   - npm run train
   - If this gives an error, run this instead:
   - python util/train.py
   - This may take a bit of time, but after it's run once, it doesn't need to be run again.

4. Launch development server using the following bash command in root of project

npm run start

The chatbot should launch.

5. (Optional) Get an API key for Google Maps Services
   - Visit Google API keys documentation
   - Make sure to get the Geocoding API, Places API, Geolocation API, and Directions API.
   - Replace the contents of googleapikey.txt with the API key.

From here down is just documentation from the last two projects, there isn't anything new to the individual assignment

Simplified Project Structure

In the individual project, there were no added files to the bot. All of the work was done within src/agent/agent.py.

There are a few new files for user setup (googleapikey.txt) and the Final Project Report though. As such, I've added them here.

```
.
├── ...
├── googleapikey.txt
├── config
│   └── dataset.json   -> Stores our dataset for NLP
├── documentation
│   ├── 30-Turn Convo.pdf   -> Stores images of the thirty-turn conversation as stipulated in requirements.
│   ├── DFD's.pdf   -> Stores the Data Flow Diagrams and descriptions of them.
│   └── Unit Test Descriptions.pdf   -> Stores descriptions of the unit tests used. ├── Project-Report-A2.pdf   -> Our project report document for A2
├── Project-Report-A3.pdf   -> Our project report document for A3
├── Project-Report-FINAL.pdf   -> A copy of my final project report for the individual assignment
├── src
│   ├── agent
│   │   ├── plugins
│   │   ├── tests
│   │   │   └── agent_test.py   -> Unit tests as used by Pytest. │   │   └── agent.py   -> The Python agent. Essentially used to read a query, manipulate it, and return
the results.
│   │   └── chat.py   -> The Python agent. Essentially used to read a query, manipulate it, and return the results.
│   ├── main
│   │   ├── nlp-service.ts    <-  Interfaces with Node NLP module and trains from dataset
│   │   ├── main.ts    <-  Electron entry point, also includes IPC module for communicating frontend
│   ├── renderer
│   │   ├── App.vue    <-  Vue entry point
│   │   ├── components    <-  Component structure following atomic design principles
│   │   │   ├── atoms    <-  Smallest component unit in atomic design
│   │   │   │   ├── ChatMessage.vue    <-  Vue component for chat messages
│   │   │   │   └── TypingMessage.vue    <-  Vue component for "user is typing..."
│   │   │   ├── molecules    <-  Medium sized component unit
│   │   │   │   ├── ChatBar.vue    <-  Vue component for chat bar
│   │   │   │   ├── ChatBox.vue    <-  Vue component for chat box (where messages go)
│   │   │   │   └── ChatHeader.vue    <-  Chat header component (recipient picture+name)
```

```
│   │   │   └── organisms    <-  Largest component unit (full functioning features)
│   │   │       └── ChatContainer.vue    <-  Chat container component (full chat feature)
│   │   ├── index.ejs    <-  Main HTML template where Vue is injected
│   │   ├── index.ts    <-  Webpack entry point
│   │   ├── models    <-  Stores reusable object models (i.e. classes/data structures)
│   │   │   ├── message.ts    <-  Class to identify a sent message
│   │   │   ├── service.ts    <-  Abstract class representing stateful services injected into Vue
│   │   │   └── user.ts   <-Class to identify users in chat
│   │   ├── services
│   │   │   ├── chat-service.ts    <-  Service to manipulate chat state and provide responses from AI
│   │   │   └── services.ts    <-  Services initializer plugin which is installed into Vue (inits all services)
│   │   ├── store    <-  Configuration for Vuex store (state management)
│   │   │   ├── createStore.ts    <-  Initializer for store (object factory)
│   │   │   └── modules    <-  Stores all modules involved in store
│   │   │       └── chat.ts    <-  Chat module for store (manages chat state for chat service)
│   │   └── util    <-  Misc utilities
│   │       ├── createApp.ts    <-  Factory to create Vue app with desired configuration
│   │       └── inject-context.ts    <-  Helper function for injecting key into Vue/components (for DI)
│   └── ...
├── util
│   ├── train.py   -> A script used to train TensorFlow.
│   └── trainer.py   -> Utility for configuration used in train.py.
```

NOTE: Not all files are included. Configuration files and similar files of low relevance (added clutter) are removed.

NOTE: Summary of Python files is very simplified.

Vue Components (pseudo classes)

ChatMessage.vue <ChatMessage>

Props (arguments):

- message -> Message (represents message object containing content/date/sender)

TypingMessage.vue <TypingMessage>

Props (arguments):

- user -> User (represents user typing)

ChatBar.vue <ChatBar>

Props (arguments): N/A

ChatBox.vue <ChatBox>

Props (arguments): N/A

ChatHeader.vue <ChatHeader>

Props (arguments):

- name -> String (represents name of user in header)

ChatContainer.vue <ChatContainer>

Props (arguments): N/A

NOTE: Prop typings are denoted by [propName] -> [type]. They correspond with native TypeScript types OR typings found in our src/models folder. HTML selectors (class names) are indicated by "[file].vue <[selector]>"

NOTE: While vue components are phyiscally represented as classes in code/memory and generally function like so, I don't necessarily know if it is the correct nomenclature. However , based upon the requirements, we can call them this.

Vue technically abides by the MVC (Model-View-Controller structure) where the View is connected two-way data bindding to the Controller (i.e. JS in Vue component). Models are represented as classes in our "src/models" folder

Models

```
├── models    <-  Stores reusable object models (i.e. classes/data structures)
│   ├── message.ts    <-  Class to identify a sent message
│   ├── service.ts    <-  Abstract class representing stateful services injected into Vue
│   └── user.ts   <-Class to identify users in chat
```

Message.ts (Message)

Class members:

- date: Date (message date sent)
- sender: User (message sender object)
- message: string (message body)

Service.ts (*abstract* Service)

Class members:

- protected app: Vue.App (pointer to Vue App instance)

User.ts (User)

Class members:

- name: string (represents user name)
- typing: boolean (represents whether user is typing, default=false)
- photo?: string (represents user photo URL, default=undefined)