Lab-4 chat-bot

Names, Majors, and Years

Luke Rowe, CSC, Grad Student

Jeremiah Lee, CSC, 5th-Year

Brandon Kwe, CSC, Grad Student

Yaniv Sagy, CSC, 4th-Year

pog-bot description and mechanics

TODO:

Phase 1

To accomplish phase 1 we started with the supplied code Dr. Foaad Gave us. With this code we modified the man loop to become a less nested, more filtered system. To accomplish this we made a helper function method named, `response_filter`. Response filter not only filters for responses that we wish to see the contents of, but also breaks up the response into four individual `sender`, `m_type`, `m_tar`, `msg`. The filter looks for messages that only the bot can read, it effectively whitelists only responses that are:

1. The `m_type` (message type) is `PRIVMSG`.
2. The `target` is the indicated channel the bot is in.
3. The `msg` starts with the botnames name with a : following right after

After the message is fully processed the following statements need only be if and elif statements asking what the `msg` contents are. Speficially for phase 1 we created all of the relevant "commands" for the bot which were all easy to implement with simple responses, however the user listing command was a little different. We created a brief helper function which will return the response of the bot issues a command to the channel asking for a list of `NAMES` formatting it and then returning it for the bot to send. This portion of the project was refined heavily and while it is very few lines, it was well thought out for the future of the project, Luke primarily laid the foundation for the start of this bot.

Phase 2

To accomplish phase 2 we went off of phase 1's format and created a bot that responds to well stuctured predefined greeting terms. It can both prompt users and be prompted by users to say hello to them and see how they are doing. There is an internal timer that the bot runs on, that takes approximately 15 seconds for it to get annoyed/try to progress the conversation. This timer was accomplished by making the recv on the socket timeout and return a dummy response containing the message dev-pass which we ignored. On timeout the code would run the timer checks and then go back to waiting. While this has the potential to miss inputs, we have yet to see such a thing as it executes the timer checks extremely quickly. It was also a lot more computationally and code efficient that making mutliple threads and locked variables.

The bot has a state system that it goes through, using the flowchart provided to us. It checks the current state and current message and if they match up then it flips through a list of predefined responses we made and chooses them relative to which part of the flowchart it is in.

Individual Reports

Luke Rowe

My contribution was having a q/a bot that answers any prompt with a `who` and `pres` in the message. It will then extract any number in the text and look for the president who corresponds with the number. From there I use BS4 to strip a wiki list of all components including `[pres_num, pres_name, pres_birth, pres_death, pres_term, pres_party, pres_election, pres_VP]`. With this data the bot extracts all info, including small little bonus things including in the wiki including `[c] == lack of political parties, [d], [g], [l] == diff party as P and VP, [e] == death in office, [f], [n] == new party, [h] = resigned, [j] == expelled from party, [t] = new VP`. This is displayed to the user in a bracketed text and was not further extrapolated due to the nature that I did not want to write 10+ if statements with new sentence parameters since I had already done that for about 5 hours total. I also implemented my own dictionary with the relative terms for first second, etc up to 99. I thought of a pretty clever NLP strategy to automate what each number correlates to.

In addition I added the function to query the bot with `tell me more about {pres name}` and any name even if shortened will respond with more info about presidents with that name, i.e. bill clinton for `bill`, or all the georges for `george`, etc.

In relation to our project I feel that navigating wiki, keying information, and reporting it all to a localized csv helps us a lot so that we can gather genelogical data easily. Additionally I went beyond the scope of the project to make sure that the bot cleans any data that we do not want, has local dev related messages it can parse through, and does not block on any given waited input to make sure that the bot progresses. I will admit a large part of the time I committed to this bot was spent on developing the bot. I have previous chat-bot experience as an employee making discord-bots so I wanted to make it up to the standards we had back then.

Jeremiah Lee

For Phase III I decided to create a redimentary language translator. The idea for this extended functionality was that you could input a command to the bot with a form similar to something like "translate this into English: Me llamo Jeremias" and the bot would respond with the translation.

To do this I wanted to utilize an translation API and began searching for one that was 1. free, and 2. could use without having to register an account with. Fortunately, Libretranslate is a free open source translation api with models in over 80 different languages. What was difficult about using libretranslate was that I couldn't use the online demo as that required a paid key, however since libretranslate is open source I can host it locally for absolutely free.

So, how this translation functioanlty actually works: first the user will input a command into the bot. The keyword to launch the translation functionality is "translate" in the bot command. If that word is present, my modules fire up.
To begin, an instance of the LibreTranslateAPI class is created. What this does is it launches libretranslate locally (defaulting to http://localhost:5000). Becuase this takes a little bit of time the **init** method also has a wait_for_server() method call to ensure that the local libretranslate is running before trying to send requests to it.
Next, the input message is parsed to extract the source and target language (if any are present) and the text to be translated. What I mean my source and target language is that in a command like "Translate this Italian into Swedish:..." Italian would be the source languague (what languauge the text is currently in) and Swedish is the target language (what languague we want to translate to).
Becuase I wanted to allow for multiple different forms of this translate command, a list of regex patterns are used to capture these source and target languauges, as well as the text to translate.
So, now the module knows what languages and what text to translate, but the libretranslate API expects the ISO language code, not the string of the Langage (de vs. German) so before the API is called a quick method is called to convert the string into the language code. This is done using a predetermined mapping with languages that libretranslate supports.
Depending on the structure of the command, the translate function of the API is called with the source and the target languages, or just the text (defaulting to the auto language detection built into libretranslate for the source and english for the target). The endpoint and params are set up and a http request is made. If it succeeds the translated text is extracted from the json response and returned as a formatted message to the user.
Finally, becuase libretranslate is hosted locally, the API kills the process running it and shutting it down until the next time someone asks the bot to translate something.

Improvements that can be made to this functioanlity is easliy more regex patterns for commands. There are only a few given that I have tested but because this is language there are lots of different ways to ask the bot to translate something. Currently I am banking on the fact that the word "translate" and optionally "this" and "into" are words commonly used in the command but other forms could be considered. Furthermore, becuase libretranslate is hosted locally there is noticable latency from issuing the command and recieving a response. Finally, more in-depth and robust language detection can be used. Currently I am relying on the automatic language detection within libretranslate, and although I considered utilizing the language detection tool from lab1 I was unconfident in its ability to detect langauge from input text that was very short, which is what this functionality is intended for.

Brandon Kwe

For Phase III, I gave our bot the ability to answer the question "When was *name* born?", *name* being the name of a famous person, historical figure, and some famous animals. This question works first by parsing the name of the person out from the IRC message. This name is then passed to the Wikipedia api call which returns the summary section of the Wikipedia article. I then use some string manipulation to strip the return string of everything but the date of birth. If the input message does not yield a valid date of birth either because the name is ambiguous or is not a person with a Wikipedia entry, the bot responds with a message indicating such: "I'm sorry, I'm not sure when *name* was born.".

Some sample names I tested with and know work:
"Xi Jinping", "Donald Trump", "Bill Burr", "Jesus Christ", "Julius Caesar", "RBG", "Mike Trout", "Winston Churchill", "Shakespeare", "Koko gorilla", "Phil Ivey"

Yaniv Sagy

For Phase III, I created an NBA data feature where the bot can respond to questions such as "NBA: Which team did [NBA team] win against on [date]?" or "NBA: Who lost to [NBA team] on [date]?". I initially wrote a Python script that used BeautifulSoup to parse a variety of NBA game data from basketball-reference.com that spanned across multiple years. I collected data such as the home team and visiting team, the points they each scored, the stadium they played in, and the amount of fans in attendance, and wrote the data to a CSV file that has been saved as nba_data.csv.

To directly answer the questions pertaining to which NBA team beat or lost to other teams on any given date, I created a module named yaniv_bot that runs a function that initially reads in the CSV file and transforms it to a pandas DataFrame. Then, when a user asks a question of the form `NBA: Who [beat/won against/defeated/lost to/lost against/fell to] [NBA team] on [date]?`, `NBA: Who did [NBA team] [beat/win against/defeat/lose to/fall to] on [date]?`, `NBA: Who did [NBA team] play against on [date]?`, or `NBA: Who played against [NBA team] on [date]?`, spacy was used to execute a dependency graph parse in order to determine the pronouns, subjects, objects, and numbers (date of the form `MM/DD/YYYY` or `MM/YYYY` or `YYYY`) in the input message. From these parts of speech and dependency relations, intuition could be built as to whether or not the noun in the input message was the subject performing the verb or the object being affected by the verb. With this knowledge, the data could be used to filter out the relevant NBA game and date rows from the pandas DataFrame, in addition to counting the number of times the team in question has won or lost games on the specified date. This data is then returned as a list of NBA game results back to the user that made the query to the bot.