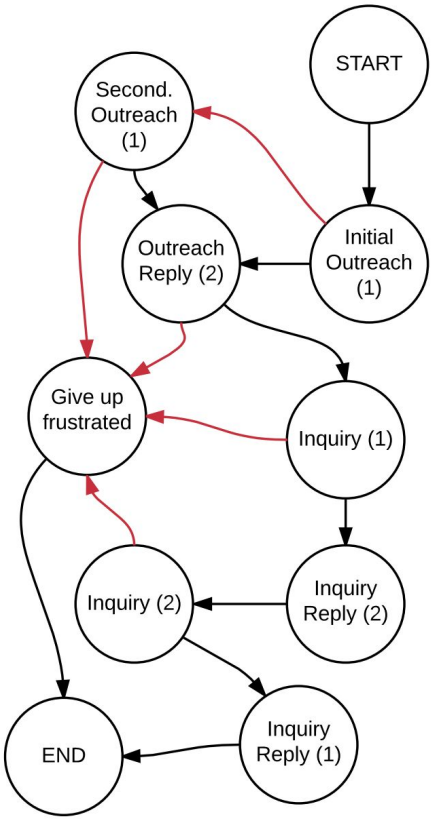
Angello Parrolivelli
Lab 4 - Chatbot!

Chatbot System, Main Loop, and Mechanics

System and Main loop

The main loop for my chatbot follows the FSM below (from the spec) extremely precisely.



One important thing to note about the chatbot before continuing that is crucial to its operation is that it sets a timer for a timeout at every state in this FSM. This includes the START state. Any contact by user to chatbot or chatbot to user resets the timer. If at any point this timer goes off, the bot will take the red path that is available at the current state. So, the chatbot starts at the START state and sets it's "reach out" timer, which determines how long it'll wait before reaching out to a random user in the room and attempting to start a conversation. If no one replies to the chatbot after an outreach, the chatbot will repeatedly timeout and go through the STATE → SECOND OUTREACH (1) → GIVE UP FRUSTRATED → END path. If someone replies to the chatbot, they become speaker 1 and the chatbot will go through the main successful loop pictured on the left. Whether the chatbot initiates conversation or not determines if it speaks at the (1) or (2) states.

This loop is powered by a `do_command()` that activates every time the chatbot is mentioned with `<chatbot-name>: <cmd>`. On every command, the chatbot advances its state in the FSM based on user input (`advance_state()`) and previous state and will reply in accordance with the next state in the FSM (`handle_action()`). If at any point "forget" is given as a command, the bot forgets everything and resets to the START state.

Mechanics

The main mechanics of changing states is handled by `handle_action()` which will choose a reply from a predefined set of replies based on what the next state in the FSM should be. Typically, this is just changing the state and choosing a reply, but I had to implement a set of conditionals throughout the function that handle special cases in Part II. For example, if we are

ever at the END state or we are the final reply before an END state, there will be an extra advancement of states / resetting state. Also, in the greeting there is a section where speaker 2 has to both reply and then make an inquiry. I have a conditional in `handle_action()` accounts for this by either not advancing the state or advancing and replying twice.

Implementation strategy, algorithms, data structures, and NLP

As mentioned before, the strategy was to emulate the FSM as closely as possible. I chose to represent the FSM as a 2-level dictionary where the initial key is a state and the value to that state is 2 key-value pairs with keys “timeout” and “success.” These represent the red and black arrows in the FSM respectively. This way, with any state change, depending on whether the chatbot has expired a timer or gotten a reply, I can just find the next state in this dictionary. I felt that this was sufficient instead of creating a Node class and Graph. States are also represented as an enumeration so that keys are not based on strings for states (better for development). Furthermore, there is a `bot_phrases` dictionary that has a set of replies for each possible state the chatbot can be in. In all, the main loop defers to `do_command()` which will check the above data structures and the status of the FSM to choose a next reply and advance the FSM.

Where things really get interesting is in the creative part. I chose to make a nutrition bot that can give you any piece of information from a nutrition label for any food OR even a summary of the main, important parts of a label for a food. First, I had to recognize if the user was asking for something from the nutrition side of the bot or the conversation part. To accomplish this, I lowercase user input and do a simple regex match for the phrases “how much,” “how many,” or “tell me about,” at the beginning of the user command. If the phrase was a “how much/many,” then I knew to get specific information for a food, and if it was “tell me about,” I was going to get general information. The next difficulty of course is finding out what 1) the food is that the user wants to inquire about and 2) what part of the nutrition label they’re interested in IF they want specific information. To accomplish this, I tokenized the user input and did POS tagging on all of the words. Because of the way these questions are framed, I just searched for all NN, NNS, or NNP in the tokenized sentence and JJ’s IF they’re not “much/many.” After this, depending on the scenario, I’ll save the specific field and then concatenate all of the other selected words after that could describe multi-word foods (ex. chocolate cake → chocolate/NN cake/NN and dark chocolate → dark/JJ chocolate/NN).

After this, but before querying the nutritional API that I found, I had to make sure the user is valid and will work with the API queries. For example, the user could have typed “sugars” or “carbohydrates” or “vitamin D” whereas the API has them as “sugar,” “carbohydrate,” and “vitamin_D” respectively. My solution to this was getting all of the fields I could possibly get form the API about a food and perform a Levenshtein distance between the user’s query word and the possible fields from the API. There’s also quite a bit of tedious string manipulation with spaces, underscores, capitalization, etc. that I had to perform to enhance the distance calculation. Furthermore, if the score is greater than 3, I warn the user that the field they want might not be available for that food.