CSC 482: Lab 5 Report

Lucas Summers (lsumme01@calpoly.edu)
Braeden Alonge (balonge@calpoly.edu)
Nathan Lim (nlim10@calpoly.edu)
Rory Smail (rsmail@calpoly.edu)

Phase I and II of the chatbot was implemented exactly to spec. The bot follows all the commands specified in the instructions, waits for a while before attempting an outreach to a random person in the server, etc. When the bot is mid conversation (after either someone has reached out to it and it has responded, OR the bot has reached out to someone else and they have responded), any greetings will be ignored until the greeting protocol is finished. However, questions for Phase III and any commands like "die", "forget", etc. will still be processed by any user even mid-conversation with another user. Another additional feature we added into the greeting protocol is a feature to automatically prompt for the 2_INQUIRY state if there was only a 2_INQUIRY_REPLY. So, for example, if a conversation started with a greeting, and the bot asked "how are you?" and the response was just "good" or something along those lines, the bot would wait for the timeout period and then give a prompt for an inquiry like "Don't you think you should ask how I'm doing?". This helps to make sure that the greeting protocol can be carried out in full. There is also a check to see if the 2_INQUIRY_REPLY is included along with the 2_INQUIRY in a single message so that the specific states don't need to be split up into two distinct messages. At the end of the protocol, as was present in the examples in the spec, the bot simply no longer sends anything. The bot will then be open to greetings from other users, but will not do any outreach of its own after completing its first one unless the "forget" command is issued.

Phase I and II were fairly simple to implement. There weren't any challenges in implementing the basic IRC connectivity and command handling. The main loop in main.py continuously polls for IRC messages using the IRC client's get_response() method, which has a 1-second timeout to avoid blocking. When messages are received, they are parsed to extract the sender, message content, and whether the message is addressed to the bot (prefixed with the bot's nickname).

For Phase II, the greeting finite state machine (FSM) was implemented in greeting_fsm.py using a class-based approach. The FSM tracks the current state, conversation partner, and timing information. States include START, 1_INITIAL_OUTREACH, 1_SECONDARY_OUTREACH, 2_OUTREACH_REPLY, 1_INQUIRY, 2_INQUIRY, 1_INQUIRY_REPLY, 2_INQUIRY_REPLY, and END (as were defined in the assignment specs). The FSM includes timeout handling (20-30 seconds) and automatically prompts for missing conversation elements in the FSM states, (like saying things like "Don't you think you should ask how I'm doing?" if a reply to the first inquiry 1_INQUIRY is received without a reciprocal inquiry).

The outreach controller (outreach_controller.py) manages all of the automated outreach timing and randomly selects a user to greet 10-20 seconds after joining the channel. It maintains a set of active channel users and updates this list both from IRC NAMES responses and any observed message activity.

Phase III was a much more complex component. Our goal was to have a system that could answer questions about country statistics even when asked in very open-ended ways. The dataset we used for this feature was sourced from Kaggle (at this link). This data is stored in a pandas DataFrame loaded from a cleaned .csv file of the data. We utilize the following fields for our feature: country, region, population, area, population density, coastline ratio, net migration, infant mortality, GDP per capita, literacy rate, phones per 1000, birthrate, and deathrate. There are a few more fields in the dataset that we decided not to use because they weren't very interesting.

The first part of our system uses spaCy's named entity recognition to identify any Geo-Political entities or Location entities within questions directed to the bot. If these types of entities were detected, we go on to the next phase of the system because we can assume that the question is about country statistics. After the NER, we use Levenshtein distance to detect spelling variations and partial matches to the countries in our database (match the entity to the country whose name has the smallest edit distance).

The second part of our system is the more impressive part. After identifying the country that the question is based off of, we use a cross-encoder model from HuggingFace (specifically "cross-encoder/ms-marco-MiniLM-L6-v2" from the sentence-transformers library) to match questions against the definitions of all of the columns. For each of the statistical categories in the dataset, a definition string is created like "Population: the total number of people living in the country". The cross-encoder computes the semantic similarity scores between the user's question and each one of the definitions, and then the category with the highest similarity score is selected for the answer. Answers are generated using template strings to format the responses (e.g. "The population of {country} is {value} people"). The cross-encoder model is run completely locally and runs very fast.

In full, the system allows for incredibly dynamically phrased questions and provides for efficient answers for varied statistical queries. For example, if you wanted to ask about the number of cellular subscriptions per 1000 people in Italy, you could phrase it in any number of ways: "How many people have phones in Italy?", "How many people are subscribed to mobile lines in Italy?", "How many people use cellular in Italy?", "In Italy, how many people have mobile devices?". All of these queries would work to correctly get the end statistic.