4/15/2022

District 93 Chatbot Project Report

Requirements

As a whole, the chatbot shall handle any user utterance and in turn provide a relevant response through a command-line interface. The chatbot shall additionally track and provide statistics of session interactions across sessions. These central requirements can be broken down and organized by the five programming assignments that form the final chatbot:

1) Program 1: This program shall extract data from the district specified and write it to a local file. This program shall also provide a line, word, and character count of the extracted data.
2) Program 2: This program shall process the extracted data from Program 1 based on questions. Given an information type as input, the program shall return the relevant content from the data. There are six main types of data available.
3) Program 3: This program shall make content available in a command-line interface. Any user response must be handled. This program shall also run infinitely until the user wants to quit.
4) Program 4: This program shall provide a method of parsing any user utterance and matching it to the chatbot's known queries. If the user utterance matches close enough to a known query, the program shall signal it.
5) Program 5: This program shall report statistics on session interactions across sessions. This program shall also record the chat between the user and the system and store it as a local file. A given chat session's local file shall also contain that session's statistics—the number of user utterances, the # of system utterances, and the time duration of the session.
6) Program 6: This program shall combine the previous five chatbots to form a fully functional chatbot with the requirements stated above.

Specification

For clarity and organization, I will discuss the decisions I made and the scope I selected per program:

1) Program 1: To make the data extraction a little more interesting, I chose to not rely on a static copy of the data. Instead, this program will first attempt to connect to the District 93 chatbot; if successful, it will retrieve the district's webpage HTML through web scraping and write it to a local text file. If the webpage cannot be reached or the web scraping fails for any other reason, the program will default to the local data that is already present. This program also takes advantage of 'pickling' by pickling the data in addition to writing to a text file.
2) Program 2: There are six main types of data that each district's webpage holds; the contact information, the personal information, and the service in public office were all readily extractable in the HTML. I chose to only extract these information types because the remaining types—committee assignments, sponsored bills in the House, and voting record—would require more than regular expressions on the HTML data to retrieve information on.
3) Program 3: I followed the requirements closely for program 3 for the most part, with additional effort placed on sanitizing the user's input—namely stripping punctuation and handling unexpected characters—and making the console output look prettier using tab formatting. To make the future chat logs (recorded with Program 5) easier to read at a glance, I had the chatbot respond to a query by repeating it back to the user first; this was also done to mimic "active listening" so that the system seemed less robotic.

4) Program 4: Instead of using the more straightforward algorithm provided in the lecture, I chose to instead use edit distance (specifically Levenshtein distance) as my metric for determining whether the user's utterance matched one of the chatbot's known queries. By putting the calculated Levenshtein distance of the two over the largest possible Levenshtein distance between them, I could obtain a confidence ratio which would directly quantify how close the user's utterance was to a known query. To make the matching even more lenient for users, I later chose to include a "partial match" state in addition to a match or a non-match; when a given utterance's ratio was within 5% of the minimum confidence required, this program would return the closest match query but specify that it was only a partial match. This led the way for my chatbot to provide suggestions.
5) Program 5: Like program 3, I followed the requirements closely. Instead of simply tracking the number of system utterances, though, I decided to track the number of "successful" utterances instead, i.e. the system responses that found information for the user's query. In my opinion, this made the tracked statistics more nuanced and useful, especially since it could provide an overall success rate across all sessions. Any suggestions made by the chatbot only contribute to the number of successful utterances if the user specifies that the suggestion was useful.
6) Program 6: This followed the requirements closely given that it was simply to merge the previous programs together.

Development Highlights

For clarity and organization, I will discuss the implementation decisions I made and the challenges I faced per program:

1) Program 1: The implementation of this program was straightforward, requiring only a single method. Originally, this program was set up with optional command-line arguments through the argparse module, but I chose to instead convert these arguments to parameters for the method itself instead so that a user could import the method directly and pass the parameters as needed. The challenge for this program was successfully handling the call to the webpage and catching any potential exceptions since many things could go wrong from reaching the website to grabbing the HTML data.
2) Program 2:
    a. I implemented much of this program's functionality as a new class called InfoContainer. As the name implies, the class contains a string of information within it; in this case, it held HTML information. These InfoContainers also hold a dictionary which helps it store and easily retrieve specific bits from the information it contains based on keywords. This is empty by default but can be assigned either during construction or using the set_dict method at runtime. I utilized inheritance by relying on several subclasses of InfoContainers to help usability and provide pre-made options for users who just wanted to import "as is". These subclasses are based on the information types we're looking for in the HTML—PersonalInfo, ContactInfo, etc.—and they build their own relevant keyword dictionaries of information. Each of these specialized subclasses has a unique dictionary-building method that uses regular expressions to pair information to keywords.
    b. Apart from debugging the necessary regular expressions needed to retrieve each piece of information, the challenging part of writing this program was making InfoContainer intuitive. I spent a good deal of time deciding how I'd like the class to work, what it

should contain, and how it ultimately interacted with the system. In the end I decided to make use of inheritance to streamline the class's use and I believe it worked well.

3) Program 3: This program relied on a single script like Program 1, but I chose to implement the functionality through multiple methods for improved readability, better modularity, and overall convenience. The centerpiece of this program is the runui method that carries out the functionality when passed the HTML info. By tying everything together into a single method with a simple parameter, using this program in the final chatbot was extremely straightforward. These helper methods completely handled calling Program 2 for information, making the console output prettier, and handling yes/no prompts. The biggest challenge I faced here was not necessarily getting the UI to run properly but deciding how I wanted to structure the rest of the project around it. In my mind there were two main options: I could have the UI as the lynchpin of the chatbot and simply call Program 4 and Program 5 from it, or I could keep it as a standalone program and somehow carry out Program 4 and Program 5's functionality completely independent of the UI. Ultimately, I chose the first option as it was the most intuitive and simplest to implement. Though the second option would be the best for pure modularity, it would have relied on a significant amount of parameter/state information passing between the UI and Program 4 and Program 5 to achieve. Lastly, a more minor challenge I had to overcome was modifying the UI to handle the newly added suggestion functionality that I chose to implement in Program 4. I had to capture and store a new boolean to properly handle this, which felt like the most straightforward and easy to understand approach.
4) Program 4: This program was my favorite to implement as it was one of the most straightforward ones to create but the most complex to fine tune. As mentioned in the 'Specification' section, I used a metric based on Levenshtein distance ratios to compare the similarity of user utterances to known queries. These ratios could then be easily viewed as a score out of 100, with 0 being completely dissimilar and 100 being a perfect match. To add to its overall robustness, I did additional research and chose to utilize a tokenized version of this method. This helped alleviate the strictness of the match by comparing the input query to the known query piece by piece; for example, using that method, the query "may I have your hat" would still perfectly match "have your hat I may". The implementation of this tokenized method was done through the "thefuzz" package, credit given in the documentation and in Program 5 itself. To further improve robustness, this program contains a synonym dictionary which automatically replaces certain words in the user's query to the ones directly used in the known queries. The challenge in this program was figuring out the right balance between the strictness and flexibility of matching. If I had set the minimum ratio too low, for instance, the user could be given a response that was completely incorrect. However, if it was too high, the chatbot would not provide relevant answers even if the user's intent did match a known query; this second circumstance is what I realized when I received my feedback for the program. I ended up addressing this by implementing a suggestion functionality. If the calculated distance ratio fell below the required amount but was still within 5% of it, this program could now pass the fact that it was a partial match to Program 3 for it to display to the user. I chose to make this partial match threshold a percentage of the required amount as it would proportionately make stricter matching easier while avoiding over-simplifying matches done with a lower ratio.
5) Program 5: I split this program into two parts to handle the variety of functionality this program needed to achieve. To log the statistics, record the user's chat, and write those details to their respective files, I created a class called a ChatSessionLogger. I designed the constructor to allow variable path locations, so a user could specify a new chat statistics csv path and the text file path if they desire. Once a ChatSessionLogger is created, you can call the class's loq_q_a method to write a user question and chatbot response to a text file. This method allows two optional

booleans for increased versatility; these allow you to specify if a chatbot response was actually helpful—which was used for my nuanced statistics, and you can specify if you'd like a newline character to be placed after question-answer in the final text file. When the chat session is over, you can use the finalize_session method to handle all the cleanup and reporting the final statistics. This method also sets an internal boolean flag which stops users of the ChatSessionLogger from logging more question-answers or finalizing the session again after finalization; a decorator called '_not_finalized' enforces this on both the log_q_a method and the finalize_session method. I think I am most proud of this class due to how little the user needs to edit their own UI to utilize it properly in their chatbot. The most challenging part of writing this class was ensuring that the statistics CSV was getting properly written to, especially since it was designed to constantly be appended. This required a completely separate helper method as well as a somewhat straightforward but a little longwinded bit of code to ensure that the header would only be created if the statistics CSV did not exist already. The second portion of this program is for reporting the statistics once they have been written to the csv and text files. I chose to handle reading in the available data and summarizing the statistics using the 'pandas' package for its DataFrame class, a popular choice for data analysis and data science. This saved a bit of effort over placing the data into a standard Python list since I was able to make use of preexisting DataFrame methods for summarizing the statistics. Using pandas also makes this program more efficient given that it relies on the numpy package under the hood, which itself is more efficient than base python for storing and handling data. Realistically, this difference in efficiency is unnoticeable as of now, but it will significantly help when you have thousands of logs to summarize and retrieve. The challenge with writing this half of Program 5 was making sure I caught every possible error when calling the methods from the command-line.

6) Program 6: Due to the heavy focus I placed on modularity and convenience in the previous programs, tying them all together in this program was simple. The challenge was successfully figuring out how to utilize the command-line portion of Program 5 given that it used argparse and required parameters to be passed from the command-line. I was able to resolve this problem with Program 1 by modifying its extract_info method to have the command-line parameters as optional parameters in the method itself, but this only worked because Program 1 was so straightforward. I ended up using Python's subprocess module from the standard library to call Program 5 from this program as if it were called by the console. This solution ended up being relatively intuitive once I understood how it worked, and it allowed the final Program 6 script to be collectively less than 20 lines of code.

Reuse

I wrote every part of this chatbot with reusability in mind, though admittedly the earlier programs could have been refactored to further improve it. In the final program, every chatbot part which relied on reading information from a specific path was refactored to be more dynamic; instead of needing to change this path in every program (or having to pass the full new paths as an argument each time), I used a very brief configuration script which automatically determines the root directory that the chatbot is being executed in. If a user wishes to change where the chatbot looks for information on a global level, they can edit this configuration script. On a more local level, users can pass more specific paths to the individual methods which make up the later programs, such as specifying a new path for writing the statistics csv in Program 5.

Reusability also heavily influenced how I went about implementing these programs. Before this class, I typically only used Python in a script executing way; in other words, the code was usually ran in one large block to carry out a specific functionality. Instead, a majority of the programs in this chatbot now utilize separate classes, and each of the classes I implemented are highly customizable (both when initializing and in runtime). This makes it way easier for users to import and utilize for their own projects versus one large method. Even the programs that did not rely on separate classes still focused on versatility, such as the UI program. The UI program allows users to directly import their own chatbot's known queries for the chatbot to parse by reading that information in from a configuration JSON file.

Lastly, the structure and intended way to use these programs were ultimately influenced by reusability. In general, each one of these programs was designed in a very "plug and play" manner. I wanted to avoid superfluous user input, imports, and method calls as much as possible, and I think I did an okay job. These programs only require the user to import one or two methods at most to function properly, and the number of necessary parameters for each are minimal, making it easier to pass results from one to the next. As a result of my efforts to improve reusability, I believe I have helped reduce the amount of work required for someone to implement parts of my district chatbot into their own.

Future Work

There are a number of things that can be done to improve my chatbot. In Program 4, it would further help reusability if the user were able to import a custom synonym dictionary much like they can with the known queries. I would also implement a more nuanced, dynamic method of matching user intent to known queries through deep learning. By using natural language processing, I could eventually set up my chatbot to actually learn and self-improve across chat sessions, similar to many existing chatbots online, which would help with maintenance. Another route for making my chatbot more useful is to increase the amount of information that it can extract from the district website. Specifically, I could have it extract the relevant information from the legislation table page and voting record in addition to the current HTML; with enough pre-processing, this could allow users to ask about specific laws and acts and how the representative voted for them. One last improvement would be to make the process of removing chat logs more convenient. As Program 5 is now, if a user removes a chat log from the chat_sessions directory, it's respective row in the chat_statistics CSV will not be removed. The program could be modified to check for and remove these missing files in the CSV when called with a certain clean-up parameter in the command-line.

Since this chatbot relies entirely on web scraping techniques for obtaining up-to-date information, Program 1 and 2 must be changed over time if the webpage's format changes. For instance, if the information on the page is stated differently, then the regular expressions must be updated. Moreover, this chatbot can only provide the latest information assuming the URL for the webpage does not change. If the entire website restructures, moves to a new domain, etc., then my chatbot will no longer be able to reach the page to obtain the HTML; this hardcoded URL must be changed over time to have the chatbot remain relevant. Apart from webpage specific problems, the code may need to be updated if the chatbot needs to be run on versions of Python which pandas or thefuzz may not support. If this is the case, Program 4 and 5 will need to be reworked to avoid using these packages, which would require the query comparison metric and statistics CSV parsing to be implemented either from scratch or from a similar package.