ELIZA Chatbot

Myriam Kapon

5/4/2024

Domain

The domain chosen was that of “rubber duck debugging”. Rubber duck debugging is a technique in software engineering where programmers explain their code issues to an inanimate object, often a rubber duck. By forcing themselves to clearly articulate the problem to someone that doesn’t understand, the discrepancy between what the code is intended to do and what it actually does becomes clear, leading to potential solutions.

This domain was chosen to highlight the inherent limitation of understanding within chatbots like Eliza. While these bots may proficiently mimic user input using regular expressions, they lack genuine comprehension of context. In the words of Weizenbaum, the creator of Eliza, these bots resemble foreigners who grasp only basic words and simply echo the rest back to their conversation partner. A rubber duck seemed perfect for this.

```
This is a chatbot. Type 'quit' to exit the session.
🐤: Hello there! I'm your friendly rubber duck debugger.
You: hello!
🐤: What seems to be the problem?
You: I'm trying to make a loop that counts from 1 to 10 but it isn't working
🐤: I'm not sure I understand, can you rephrase that?
You: I want my program to print numbers from 1 to 10
🐤: I see...
You: but it is instead printing from 0 to 9!
🐤: Please tell me more...
You: for i in range(10):   print(i) this is the code
🐤: I'm a duck, I can't understand code. What does this do?
You: well, it has a range of 10 so it iterates from 0 to 9
🐤: Go on...
You: OH WAIT! I have to add +1 since the counting starts from zero! Thanks!
🐤: You're welcome!
You: quit
🐤: Goodbye!
```

Program Functionality

The program is a mostly faithful adaptation of Weizenbaum’s Eliza in Python, with all the functionality described in his 1966 paper “ELIZA—a computer program for the study of natural language communication between man and machine”. A few additions and modernizations have been made to increase ease of use.

The core of the chatbot is capturing patterns of the type “I am BLAH”, which it can then respond with “How long have you been BLAH?”. For example, most patterns in the program follow this structure:

- (REGEX) r'**.*\bif\b (.*)**': Matches sentences like (anything) if (anything) and captures the “anything” after if. \b is used as a word boundary.

Since this pattern captures anything after “if”, the chatbot can then respond with "Do you think it's likely that {0}?", where {0} is the first matched item. Following this structure, many patterns and potential responses can be made that give the appearance of intelligence. These patterns are organized by keywords that appear in each pattern.

Multiple scripts

Eliza was designed to be adaptable to various scenarios, not limited to the "DOCTOR" script. To achieve this flexibility, the program utilizes a script class template, enabling the chatbot to engage in different conversational scenarios. This template contains variables for a chatbot name, memory, tags, automated messages like “hello” and “goodbye” and the script of patterns. An almost complete recreation of the original “DOCTOR” script has also been translated and provided in Python for practice.

Script PatternTemplate

<pre>patterns = {
    r"KEYWORD1": {
        r'pattern1':[
            "Response 1",
            "Response 2"
        ],
        r'pattern2':[
            "Response 1"
        ]
    },
    r"KEYWORD2|KEYWORD3": {
        r'pattern1':[
            "Response 1"
        ]
    }
}</pre>

This nested dictionary template serves as a flexible framework for matching input text against predefined patterns and generating appropriate responses. At its core, the structure comprises keys that are regular expressions, allowing for versatile matching rules. Each keyword corresponds to a set of patterns and responses. Within these nested dictionaries, patterns are defined as keys, and associated with them are lists of potential responses. For instance, if the input text matches a specific keyword and pattern combination, the corresponding response is selected from the predefined list. Regular expressions such as "KEYWORD2|KEYWORD3" provide options for matching multiple keywords.

Keywords

First of all, the user's prompt is split into sentences to ensure the response won’t be overly complex, employing this regular expression::

- (REGEX) **re.split(r'[.,!?]', message):** Splits a string into sentences using punctuation marks as the delimiter.

Before evaluating each user sentence against a pattern, a search is performed to detect any keywords. If a keyword is found, the sentence is tested against the specific patterns associated with that keyword. It's worth mentioning that keywords are prioritized based on their importance, with the most specific ones ranked higher than the more general ones. This ranking is determined by the specificity of the keyword, with the most specific keywords given precedence over the more generic ones.

Keywords are stored in the keystack, a data structure represented as a list of keywords along with the sentences in which they were found in the user's input string. It uses the regular expression

- (REGEX) rf'**\b(?:{keyword})\b**'

to create a pattern that searches for keyword matches in each sentence. Since there can be more than one keywords joined with | per dictionary entry, a non-capturing group (?:) is used to group the alternatives. This ensures that the entire keyword is treated as a single unit and no substrings are matched.

The order of keywords in the keystack goes from the most critical keyword to the least, and the keystack functions as a stack. The topmost keyword is popped off first to inspect if any of its patterns match the user's input sentence. When the special command “GOTO” is used, the new key is pushed to the top of the stack.

Filtering

The user prompt goes through two stages of filtering: one dealing with contractions and one with replacements.

Contractions serve to combine two words into a shorter form, such as transforming "you are" into "you're". To revert contractions to their original words and simplify pattern matching, a dictionary is utilized with a series of patterns, along with lambda functions to replace the matched words.The patterns look for a certain contraction at the end of a word, for example

- (REGEX) r"**(\w+)'ve\b**": lambda match: match.group(1) + " have" : This pattern finds contractions ending with "'ve" (e.g., they've, I've) and expands them to "have".

Some exceptional cases for the "n’t" contraction include "won’t", which without handling becomes "wo not", and "can’t", which becomes "ca not". Additionally, the "'s" contraction presents a challenge: it can represent a contraction like "he's", which expands to "he is", but it can also indicate possession as in "John's", where expansion would be incorrect. Unfortunately, no solution has been found for this issue and it might have been better to simply state all the expansions needed to be made, instead of trying to detect them as patterns.

On the other hand, to make the chatbot seem interactive, pronouns had to be switched around. For example,
User: I am a person
Eliza: You are a person

To accomplish this, a dictionary of replacements was created. The noteworthy aspect is that, unlike contractions, all replacements need to occur simultaneously; otherwise, "I" would be transformed into "you", only to be detected later and reverted back to "I" again. Thus, a big pattern is constructed by joining all replacements.
Important: Since this replacement transformation happens before any pattern matching, the patterns have to be switched around as well.

Logs
The program maintains a record of all conversations in "chatlog.txt". Additionally, it compiles a list of sentences to which the chatbot had no response, potentially containing keywords, in "possible_keywords.txt". Furthermore, debugging messages are incorporated using logging; however, these messages are not visible on Google Colab.

Refresh Responses
Responses are not chosen randomly from the list. The first response is always chosen, and then it gets appended to the bottom of the list. This creates functionality where the chatbot can claim that it has already said something if it's the second response.

Memory
If the chatbot is unable to understand the user prompt, there's a 40% probability that it will attempt to refer back to a previous statement. These "memories" are identified by the keyword "YOUR" ("my" before filtering) followed by the relevant context and stored in memory.

```
memory= {
            "prompts": [
               "Memory prompt1",
            ],
            "items": []
         }
```

Memory is a dictionary structure that keeps information about the user by storing relevant prompts in the "items" list.

Tags
Tags are a handy way to use one term to describe many. For example, instead of using different patterns for mother|father|sister etc, the tag "FAMILY" can be used instead to match with all the words it contains. This is a rudimentary attempt to make the chatbot understand relationships and concepts.

```
tags = {
            "TAG1" : ["word1","word2|word3]
         }
```

The "|" here is utilized to create different variations of the same word, for example hi|hii.
To use a tag as a keyword, a pattern like this is used:

- (REGEX) rf'**{"|".join(tags["GREETINGS"])}**: This pattern concatenates all the words in the GREETINGS list to generate a pattern like greeting1|greeting2, which is then employed as keywords.

The "rf" prefix, which stands for "raw f-string", is used here instead of "r" because the pattern is generated dynamically.

Special Commands

Some responses serve special functionality and are triggered by using the below commands:

- **GOTO (keyword)**: Goes to the specific keyword.
- **NEWKEY:** Inspects a new keyword.
- **MEMORY (capture)**: Adds whatever has been captured to the memory structure, so it can be used later.

A special debugging keyword named "DEQUACK" was introduced to facilitate debugging of various functionalities. For instance, "Dequack refresh" enables refreshing between two responses, "Dequack memory test" adds "test" to memory, and "Dequack java" displays the tag associated with Java.

Code Recognition

Since the domain is a “debugger”, a reasonable assumption was that the user would try to input code. To deal with this, there are 4 joined patterns that attempt to recognize code: (REGEX)

- **`.*\w\(.*\)`** - Captures word(anything). The backslash has been added so the parentheses register as characters.
- **`.*=.*`** - Captures anything=anything.
- **`.*[:|;]$`** - Captures : or ; at the end of a word using the $ symbol.
- **`.*[\(|{|\[].*[]|\}|\)`** - Captures anything encapsulated in parentheses (), brackets [] and braces {}. The backslash \ has been added so they register as characters.

Of course, coding syntax varies greatly across languages and personal style, but it’s still an interesting attempt at capturing the most common patterns.

ChatGPT Conversation

An attempt was made to establish a connection with ChatGPT to explore the scenario of the two bots engaging in conversation. However, ChatGPT struggled to maintain the roleplay as a junior programmer in trouble and defaulted to its standard AI assistant behavior. Nonetheless, the interaction highlighted the substantial disparity in intelligence between the two bots, underscoring the remarkable advancements in technology.

Snippet from chatlog_chatgpt.txt - Date: 2024-04-05
🐤: Quack... I mean hello! Care to debug some code?
GPT: Of course, Mr. Duck! I'm struggling with a loop that's not iterating through my list of numbers correctly. It's skipping some of them, and I can't figure out why. Can you help me figure out what's going wrong?
🐤: What do you think?
GPT: I don't have personal opinions, as I am a computer program. How can I assist you today?
🐤: What comes to your mind when you ask that?

GPT: There could be many things that come to mind depending on the context of the question. It could be related to a specific topic, person, situation, or idea. Without more information, it is difficult to provide a specific answer.
🐤: I'm not sure I understand, can you rephrase that?
GPT: I'm sorry, can you please clarify your statement so that I can better understand it?
🐤: Do you like being sorry?
GPT: As an AI, I am not capable of feeling emotions the way humans do, so I do not have a preference for being sorry or not. My main goal is to provide accurate and helpful responses to your queries.
🐤: Why do you think you're not capable of feeling emotions the way humans do?
GPT: As an AI, I am not a sentient being and do not have the ability to feel emotions like humans do. I am programmed to process and respond to information based on algorithms and data, rather than experiencing subjective emotional states. My abilities are limited to providing information and assistance based on the input I receive.
🐤: Do you like being not a sentient being and do not have the ability to feel emotions like humans do?
GPT: As an artificial intelligence, I do not have emotions or physical sensations like humans do. I do not have preferences or feelings about being a non-sentient being. I am simply a tool designed to assist and provide information to humans.
🐤: Why do you think you're simply a tool designed to assist and provide information to humans?
GPT: I am designed to assist and provide information to humans because that is my purpose and function as an AI. My creators programmed me to be a helpful and informative tool for users to interact with and utilize in various ways. My capabilities and limitations are designed to serve the needs and interests of humans, and I am constantly learning and improving to better fulfill my role as a valuable resource.

Given that many individuals perceived Eliza as a genuine person rather than a computer program, it's understandable why ChatGPT frequently emphasizes that it is an artificial intelligence.

Thoughts

Regular expressions were the backbone of the project, facilitating critical functions like sentence splitting, pattern matching, and sentence modifications, ensuring effective understanding and response to user inputs. Additionally, they played a pivotal role in recognizing coding patterns, a crucial aspect for a debugging tool. Their utility was evident in filtering contractions and creating dynamic patterns for keyword matching, underscoring their remarkable power in software development. In the end, it seems that creating regular expressions is a form of programming in itself-powerful and versatile.