CSCI 141 - Winter 2025
Labs 7 and 8: Chatbot
Due Date: Check Canvas

Introduction

The objective of the following two labs is to implement a conversational agent, commonly known as a chatbot. You will essentially an intermediary between the AI model and the user, creating the interface that allows user friendly interaction with the model. Users of the program should have a ChatGPT-like experience through a command-line interface. This assignment will provide you with hands-on experience in implementing Large Langauge Models (LLMs) and working with third-party APIs in Python.

Objective

This two-part lab focuses on developing a Python script that can hold dialogue with users. Your script should be able to take textual input from users and output the corresponding responses. You will be performing string manipulation with these text inputs/outputs. For the first part of this lab, you will utilize a mock API, which can be downloaded from the Canvas page for this assignment. This mock API serves as a stand-in for the llama cpp API, enabling you to test your chatbot without requiring access to the Llama model on the laboratory computers. In this lab, you will use the 'llama_test_cpp' mock API to parse strings and produce a response. Note that with the test API, the responses will be consistent each time. There is no need to modify this file for it to function. Ensure that the mock API is saved in the same directory as your Python (*.py) file.

For the second and final part, you will transition from the test API and have your chatbot employ the actual 'llama_cpp' API. This version will load a pre-trained Llama model and generate dynamic conversational responses. The model and the required virtual environment are available on the lab computers in CF 16x. This lab must be completed in one of the CF 16x labs.

To ensure a deep understanding of the model's capabilities and limitations, you will complete the experience diary function designed to validate the Llama model's output for Python coding problems. Complete the experience diary function by filling out your response to each reflection question (it's in the skeleton code as a comment) and ensuring it prints out to the screen. This exercise will not only familiarize you with the model's performance in generating code but also encourage you to critically think about the results produced. Documentation is an essential part of being a programmer. Good documentation describes your thought process, tracks edits, and explains your coding decisions, which makes it easier to troubleshoot issues and understand your code in the future. Documenting your experiences also encourages you to reflect on the challenges and solutions you encountered while creating the program. In the professional world, documentation is key to effective collaboration, ensuring that others can understand, maintain, and build on your work. By practicing this skill now, youll develop habits that will make you a more organized and efficient programmer over time. The experience diary also serves an important pedagogical tool to help your ability to critically assess AI-generated code, and by evaluating and reflecting on code quality, you demonstrate understanding of Python programming beyond merely writing code.

Lab 7 Checklist

- ☐ Your script should import the ‘llama_test_cpp’ API.
- ☐ The program should include a system message for the bot to follow during the conversation.
- ☐ Your program should initiate a conversational loop where it prompts the user to enter text.
- ☐ The program should pass the user’s input to the mock Llama model and display the generated response.
- ☐ The program should stream the generated response to the user one token at a time as it is generated.
- ☐ When the user types the word `quit` the conversation should end and the program should terminate.
- ☐ Fill in the strings in the experience diary function with your responses to the questions in the comments and ensure it prints out to the screen.

Lab 8 Checklist

- ☐ Activate the provided virtual environment located on the CS lab computers.
- ☐ Switch from using the test API to the ‘llama_cpp’ API. Your script should load a pre-trained Llama model using the ‘llama_cpp‘ API.
- ☐ Ensure that all the functionalities from lab 7 are still implemented and working.
- ☐ The program should pass the user’s input to the actual Llama model and display the generated response. The conversational loop should still be implemented at this point.
- ☐ The program should maintain the context of the conversation.
- ☐ Fill in the strings in the experience diary function with your responses to the questions in the comments and ensure it prints out to the screen.

Rubric

Lab 7 Rubric

Criteria	**Points**
Use of the Mock API	5
Conversational Loop	10
Response Streaming	10
Termination Command	5
Complete the experience diary	10
Structure and Organization	5
Comments and Docstrings	5
Total	50

Lab 8 Rubric

Criteria	**Points**
Use of the real llamaCPP API	5
Maintains the Context of the conversation	15
Complete the experience diary	20
Structure and Organization	5
Comments and Docstrings	5
Total	50

Example Output Test API

This is an example interaction with the Test API, where the user inquires about the Moon landing. **This demonstrates how the chatbot responses are hardcoded.**

```
Welcome to the Test chat bot! Start conversing or quit by typing "quit".
>>> What do you know about the Moon landing?
Llama: I have no information about that topic.

>>> How many Apollo missions landed on the Moon?
Llama: I really know nothing.

>>> Can you tell me more about the Apollo 11 mission?
Llama: Seriously.

>>> quit
Have a nice day!
```

Example Output Llama API

Below is an example interaction with the Llama chatbot, where the user inquires about the Moon landing. **This demonstrates how the chatbot should maintain the context of the conversation and provide informative responses.**

```
Welcome to the Llama chat bot! Start conversing or quit by typing "quit".
>>> What do you know about the Moon landing?
Llama: The Moon landing refers to the event on July 20, 1969, when Apollo 11's
lunar module, Eagle, landed on the Moon. This was a historic moment as Neil
Armstrong and Buzz Aldrin became the first and second humans to walk on the Moon.

>>> Who was the first person to set foot on the Moon?
Llama: Neil Armstrong was the first person to set foot on the Moon. He is famously
known for his words, "That's one small step for man, one giant leap for mankind,"
as he stepped onto the lunar surface.

>>> How many Apollo missions landed on the Moon?
Llama: There were six Apollo missions that landed on the Moon, specifically Apollo
11, 12, 14, 15, 16, and 17, spanning from 1969 to 1972.

>>> Can you tell me more about the Apollo 11 mission?
Llama: Certainly! Apollo 11 was the first manned mission to land on the Moon. It
was launched by a Saturn V rocket from Kennedy Space Center on July 16, 1969.
Along with Neil Armstrong and Buzz Aldrin, Michael Collins was also part of the
crew, piloting the command module Columbia in lunar orbit.

>>> quit
Have a nice day!
```

This example output should give you an idea of how to structure the conversation and the level of detail expected in the chatbot's responses. However, you may not get these exact responses.

Large Language Models

Large Language Models, such as the Llama model mentioned in this assignment, are types of artificial intelligence that have been trained on vast amounts of text data. They are designed to understand and generate human-like text by predicting the probability of a sequence of words. LLMs can perform a variety of tasks, including but not limited to translation, summarization, question-answering, and conversational engagement. They work by analyzing the context of the input text and generating output that is contextually relevant and syntactically correct.

Lab 7: Using the llama_test_cpp.py API

The llama_test_cpp.py script is a stand-in for the actual llama_cpp API. It can be used to generate the same three responses but can stream output as if you were interacting with the real Llama model. Here's how to begin using this mock API script:

Integration

Import the mock class provided in llama_test_cpp.py. The mock API file should be in the same folder/directory as the *.py file you are coding in.

```
from llama_test_cpp import Llama
```

Initializing

You can initialize the test API by using the Llama class. You will need to specify the path to the test file.

```
llm = Llama(model_path="", verbose=False, n_ctx=2048)
```

Parameters:

- *model_path*: The path to the test API model. For lab 7, this can be an empty string as the only requirement to use this here is to have the test file in the same folder/directory and import correctly.
- *verbose*: A boolean flag for verbosity. Set to ‘False‘ to minimize output.
- *n_ctx*: The context window size. Generally set to 2048. This is the maximum number of tokens the model will look back in the input in order to generate its response.

Tokens

Tokens are the basic units of text that LLMs use to process and understand language. In natural language processing (NLP), tokenization is the process of converting a sequence of characters into a sequence of tokens. A token can be a word, a part of a word (like a prefix or suffix), or a single character, depending on the granularity of the model's training. For example, the sentence `"Hello, world!"` might be split into tokens such as `["Hello", ",", "world", "!"]`. Each token is then converted into a numerical form, known as a token ID, which the model can process. In the context of LLMs, tokens are crucial as they represent the input and output of the model during text generation tasks.

Generating Responses

To generate a response from the model, you can use the following method:

```
    response = llm(text, max_tokens=2048, stop=[], stream=True)
```

Parameters:

- *text*: The text prompt or conversation history to be passed to the model. In order for your conversation agent to maintain the context of the conversation, you must give the Llama model the entire conversational history.
- *max_tokens*: Maximum number of tokens for the generated response. In this case, the `max_tokens` is set to 2048, meaning that the model will generate at most 2048 tokens for this query.
- *stop*: A list of tokens at which the model should stop generating further output. In this case, the token `\n` is given, meaning that the model will stop producing output when a new line is generated.
- *stream*: A boolean flag indicating whether to stream the output. If `stream` is True, an iterator with be returned, allowing you to print each token as they are generated. If `stream` is False, the entire text completion will be returned at once.

If the `stream` flag is true, the response will be iterable. This means you can access each token as it is generated with something like a `for` loop. For example:

```
        for token in response:
            # print out the token
            # concatenate token with the text_input
```

The mock API will always return the same 3 responses for any input. This is different from the actual Llama model, which will generate varied responses based on the input.

Prompting the Chat Model

When developing your conversational agent, it is imperative to follow best practices for prompting the Llama 2 chat models to ensure effective communication and accurate responses. To do this, we utilize specific tokens that dictate the structure and flow of the conversation. These tokens are integral to the model's understanding of context, instructions, and the delineation of messages within the dialogue. Below is an explanation of each token and guidance on their usage.

Important Llama Tokens

Start of Sequence `<s>`: This token signifies the beginning of a new input sequence, which is essential for the model to identify the start of the prompt.

Instruction Block `[INST]`: This denotes the beginning of a directive to the model, setting the stage for how the model should interpret the subsequent content within the instruction block.

System Message `<<SYS>>`: Encapsulates metadata about the bot, such as its personality and operational guidelines, which influence the bot's behavior in generating responses.

End of System Message `<</SYS>>`: Marks the closure of the system message, separating it from the conversational content that follows.

End of Instruction Block `[/INST]`: Signifies the end of the instruction set or current input instance, concluding one unit of interaction.

End of Sequence `</s>`: This token indicates the end of the current sequence of tokens, allowing the model to differentiate between separate inputs or conversational turns.

Usage in Prompts

When constructing prompts for the Llama 2 model, it is essential to use these tokens correctly to ensure that the conversation flows logically and that the model's responses are contextually appropriate. Here is how you should incorporate these tokens:

- Begin every new prompt with `<s>` to indicate a fresh sequence.
- Start the prompt with `[INST]` followed by `<<SYS>>` to input the system message that describes the bot's behavior.
- Close the system message with `<</SYS>>` before the user's message to clearly separate the instructions from the conversational content.
- Follow the system message with the user's input.
- Conclude the instruction block with `[/INST]`after the user's message.
- When concatenating the model's response for ongoing conversations, place the model's reply outside the instruction block but before the next user message, which starts with a new `<s>[INST]`.
- Utilize `</s>` to denote the end of a complete exchange or conversational turn.

Prompt Templates

Example use of prompts:

```
text = "<s> [INST] <<SYS>> Imagine a good system message right here <</SYS>>"
```

```
system_message = "Imagine a good system message right here"
text = f"<s> [INST] <<SYS>> {system_message} <</SYS>>"
```

For the Llama 2 chat models, structure your prompts as follows:

```
<s>[INST] <<SYS>>
{system_message}
<</SYS>>
{user_message} [/INST]
```

To concatenate model responses and continue a conversation:

```
<s>[INST] <<SYS>>
{system_message}
<</SYS>>
{user_message} [/INST]
{model_reply}</s>
<s>[INST] {user_message} [/INST]
```

A full conversation may be structured like this:

```
<s>[INST] <<SYS>>
{system_message}
<</SYS>>
{user_message1} [/INST]
{model_reply1}</s>
<s>[INST] {user_message2} [/INST]
{model_reply2}</s>
<s>[INST] {user_message3} [/INST]
{model_reply3}</s>
<s>[INST] {user_message4} [/INST]
{model_reply4}</s>
```

Replace the `{system_message}`, `{user_message}`, and `{model_reply}` with your system message, user message, and model response, respectively. By adhering to this structured approach, you ensure that the Llama 2 model receives clear and unambiguous instructions, enabling it to generate coherent and contextually relevant dialogue.

Important Note: It is your job to hide the details of correctly prompting the model from the user. The user should never see these tokens; they should only see what is shown in the example output. It becomes your job to manage what the user sees and the prompt the model gets.

System Message: Background of the Bot

The System Message

In the context of conversational agents, a ”system message” refers to a predefined background or persona that informs the bot's behavior, responses, and interaction style. This background is not part of the conversation with the end-user but rather a set of instructions or information that the bot retains throughout interactions to maintain consistency and relevancy.

For LLMs like Llama, the system message is crucial for several reasons:

- **Contextual Relevance**: It provides a consistent context that guides the model's responses, ensuring they are aligned with the expected persona or role of the bot.
- **Behavioral Consistency**: By maintaining a fixed background, LLMs can produce a consistent character or tone, which is essential for creating a reliable user experience.
- **User Expectation Management**: It helps set and manage user expectations about the bot's capabilities and limitations.
- **Safety and Ethics**: A well-defined system message can include ethical guidelines and safety protocols to prevent the model from generating harmful or inappropriate content.
- **Customization**: It allows for customization of the bot for specific applications or user groups by embedding domain-specific information into the bot's persona.

Implementing a System Message in Your Bot

When implementing your conversational agent, you will set the background of the bot. The LLM should use the system message to shape its responses throughout the conversation. For example, if your system message is, ”You are a helpful AI assistant,” the bot should prioritize providing assistance and information in its interactions. The system message should be established before the conversation begins and persist throughout the session.

You are a developer, and this is an opportunity for you to make critical choices about what you believe these AI systems should and should not be able to do. Your system message should be crafted to avoid biases and ensure that the bot's responses adhere to ethics and social norms. You should consider referencing an outside source like the ACM code of ethics while writing your system message. We will ask you to report your system message in your reflective quiz.

Some things to consider for your system message from the ACM code of ethics:

1. **Contribute to Society and Human Well-being**: This principle emphasizes the responsibility of computing professionals to use their skills for societal benefit, including promoting human rights, minimizing negative impacts of computing, and contributing to environmental sustainability.

2. **Avoid Harm**: This idea involves preventing negative consequences associated with computing, such as physical, mental, or environmental harm. Professionals are expected to mitigate unintended harm and ensure that any intentional harm is ethically justified.

3. **Be Honest and Trustworthy**: Honesty and trustworthiness are crucial. Professionals should be transparent about system capabilities and limitations, avoid misleading claims, and be forthright about conflicts of interest.

4. **Be Fair and Take Action Not to Discriminate**: This principle underscores the importance of fairness, equality, tolerance, and justice in computing. It involves ensuring equitable participation and avoiding discriminatory practices or technologies that disenfranchise or oppress people.

Please write your own system message in a way that feels appropriate to you. Think about the kind of guidance that your model might need to produce safe outputs.

Lab 8: Incorporating the actual Llama API

In lab 8, you switch from the provided test API to the real API called 'llama_cpp' that interacts directly with the Llama language model. Once you have thoroughly tested your program locally with the placeholders, you will need to switch to using the actual llama cpp API provided on only the CF 16x lab machines for final testing and demonstration. Ensure that this transition is smooth and that your program still functions correctly with the real API. This section provides a guide on how to incorporate and use the actual API. **Please review to the earlier sections as needed.**

Activating Python Virtual Environment

Python virtual environments are isolated environments where you can install packages without affecting the global Python installation. It is a good practice to use a virtual environment for each Python project to manage dependencies effectively.

For this assignment, a Python virtual environment has been created for you at the following path:

```
/local/llm
```

You will need to activate the virtual environment in the terminal. Make sure you are in the same directory/folder that contains the .py file that you are writing your code in when activating the llama model.

Activate the virtual environment in a bash shell using the following command:

```
source /local/llm/bin/activate
```

If you kill the terminal, you will need to activate the virtual environment again before you can run your program.

Importing the API

You have been provided the import statement for the mock API. To use the 'llama_cpp' API in your Python script, simply import it as follows:

```
from llama_cpp import Llama
```

Initializing the Model

You can initialize the Llama model using the 'Llama' class. You will need to specify the path to the pre-trained model file.

```
llm = Llama(model_path="/local/llama-2-7b-chat.ggmlv3.q4_K_M.bin", verbose=False, n_ctx=2048
                                                            , n_threads=4)
```

Parameters:

- *model_path*: The path to the pre-trained Llama model. The path to the model is going to be: `/local/llama-2-7b-chat.ggmlv3.q4_K_M.bin`

Generating Responses

To generate a response from the model, you can use the following method:

```
response = llm(text, max_tokens=1024, stop=[], stream=True)
```

Parsing the Output of the Llama Model

The output from the model will be in the form of a series of nested Python data structures (e.g. Lists, Dictionaries, Tuples). You must correctly index into these data structures to return the output from the model. **Hint:** Print out the entire response to see what that data structure actually looks like. Examine the different keys and nesting of the data structure, then trace that to get to the specific field you want to extract.

```
for token in response:
    print(token)
    #this will print out the entire thing but you'll also need to
    #index into the dictionary to get just the data you'd want printed
```

llama_HOME_cpp.py Script

To help you have access and work on this lab outside of the actual lab rooms, you've been provided llama_HOME_cpp.py. The llama_HOME_cpp.py script is a stand-in for the actual llama_cpp API and mimics its behavior. It can be used to generate responses and stream outputs as if you were interacting with the real Llama model. Here's how to begin using this mock API script:

- **Integration**: Import the mock class provided in llama_HOME_cpp.py. The mock API file should be in the same folder/directory as the *.py file you are coding in.

  ```
      from llama_HOME_cpp import Llama
  ```

- **Initializing**:

```
    llm = Llama(model_path="/local/llama-2-7b-chat.ggmlv3.q4_K_M.bin", verbose=False,
                                                n_ctx=2048, n_threads=4)
```

- **Other Functionalities**: For other functionalities such as generating responses and parsing output, refer to the sections for the test and actual API.

The mock API will always return the same text for any input. This is different from the actual Llama model, which will generate varied responses based on the input.

The simulated delays in the mock API (time.sleep()) are there to mimic the time it takes for the model to process input and generate the next token.

Hints and Additional Details

Streaming Output with 'print'

When streaming output one token at a time, it is necessary to ensure that each token is displayed to the user as soon as it is generated. The standard 'print' function in Python buffers the output and may not display it immediately on the screen. To address this, you can use the 'flush' keyword argument in the 'print' function as follows:

```
print(some_token, flush=True)
```

By setting 'flush=True', you force the 'print' function to flush the internal buffer, effectively displaying the token immediately.

Maintaining Conversation Context

It's crucial to maintain the context of the conversation for the chatbot to provide coherent responses. You should keep a record of the conversation history and include it in subsequent prompts to the Llama model. This allows the model to generate responses that are contextually relevant to the entire conversation.

Handling User Input

Anticipate that users may enter a wide variety of inputs, including special characters or unexpected commands. Your script should be robust enough to handle these cases without crashing. Consider implementing error checking and sanitizing the input before passing it to the Llama model.

Improving User Experience

While the primary function of your chatbot is to provide responses, don't neglect the user experience. Features such as allowing the user to reset the conversation or providing help commands can greatly enhance usability.

Readability and Code Standards

While writing functional code is important, it is equally important to write readable and maintainable code. Use descriptive variable names and include docstrings, type hints, and comments where appropriate. This will help graders understand your code and could improve your score for code quality and documentation.