VIETNAMESE GERMAN UNIVERSITY

COMPULSORY ELECTIVE 2

Chatbot Project

Author:
10421011 - Le Dang Nhat Duong
10421042 - Nguyen Ngoc Hoang Nam

Supervisor:
Dr. Dinh Quang Vinh

Group Name: Chatbot
Computer Science and Engineering

January 10, 2025

VIETNAMESE GERMAN UNIVERSITY

Abstract

Faculty Name
Computer Science and Engineering

Bachelor of Computer Science

Chatbot Project

by Nguyen Ngoc Hoang Nam
Le Dang Nhat Duong

Dr. Dinh Quang Vinh's Compulsory Elective 2 project focuses on how to apply programming, databases to a build a AI Chatbot. The course involves some algorithms and useful technique to help us build a good foundation on Machine Learning.

Acknowledgements

We would like to express our sincere gratitude to Dr. Dinh Quang Vinh for your unflinching support during the entire process of bringing our Chatbot project to this historic milestone. This crucial assistance played a pivotal role in facilitating the creation of our project. We are sincerely grateful for your unwavering commitment and guidance during this transformative endeavor.

Contents

Chapter 1

Introduction

Creating a chatbot project introduces a dynamic way to automate responses and enhance interactions between users and systems. Chatbots are computer programs that simulate human-like conversations through text or voice, commonly utilized in customer service, e-commerce, social media, and various business applications. This project aims to develop a chatbot capable of engaging users, answering queries, and providing relevant assistance based on predefined knowledge or learned behaviors.

1.1 Project Topic

Our project is building a context-augmented chatbot using a Data Agent. This agent, powered by LLMs, is capable of intelligently executing tasks over your data. The end result is a chatbot agent equipped with a robust set of data interface tools provided by LlamaIndex to answer queries about our data.

1.2 Team Members and Roles

The project team comprises members with designated roles to ensure a comprehensive development process:

- **Le Dang Nhat Duong**: Responsible for gathering data, writing a report.
- **Nguyen Ngoc Hoang Nam**: Focuses on creating a chatbot.

1.3 Tools and Platforms

To bring this project to life, we utilized a range of tools and platforms tailored to our development needs:

- **Python**: Python 3.8 or higher
- **Core AI Model (LLM)**: Open AI-GPT 4 can be accessed through OpenAI's API and offer powerful language understanding and generation capabilities.
- **Chainlit Interface**: It includes built-in Chainlit Framework, Interactive UI, Custom Components.

- **OpenAI**: It includes built-in Chainlit Framework, Interactive UI, Custom Components.
- **Google OAUTH**: It includes built-in Chainlit Framework, Interactive UI, Custom Components.

By leveraging these technologies and dividing responsibilities among team members, we aimed to create a fully functional and user-friendly chatbot platform. The project not only serves as a practical application of our academic knowledge in programming, software engineering, databases, and distributed systems but also provides a valuable real-world solution for companies.

Chapter 2

Project Requirements and Overview

2.1 Key Features

1. **Topic-based Q and A**
   - Enable the chatbot to answer queries accurately on a chosen topic.
2. **Chainlit UI**
   - Design an intuitive UI for user interactions.
3. **Memory for Contextual QA**
   - Add functionality for tracking conversation history, allowing the chatbot to retain context.
4. **Locally Deployment**
   - Host the chatbot on a locally service to ensure scalability and accessibility.

2.2 Chatbot's preparation

Our chatbot's aims is to support customer with Question And Answer in economy aspect.

1. **Data**
   - We have 20 html file to collect data from Wikipedia
2. **Dependencies**
   - **Chainlit**: A framework that simplifies the creation of chatbot applications with a ChatGPT-like interface.
   - **OpenAI API**: In this projectt, we use OpenAI model "gpt-4o-mini".
   - **Google OAUTH**: A framework from Google that allows us to validate user's identity.

Chapter 3

Building the Chainlit Chatbot

This chapter provides a comprehensive overview of the Chainlit UI for the Chatbot.

3.1 Overview of Chainlit for Chatbot Interfaces

Setting up Chainlit

1. **Environment Setup**

   - **Create Environment**:

```
pip install chainlit
pip install llama_index
pip install authlib
pip install -U unstructured
pip install python-magic-bin==0.4.14
```

Chainlit Implementation

To create a chatbot, firstly, we need a tool to train it. We choose "OpenAI 4o mini" to be that tool for thoose reasons: the model is good enough for us to train data, GPT is one of the most popular and powerful AI in the internet and it is very affordable for student like us.

2. **Step 1**

   - **Get API key**: We go to their website and buy the "OpenAI 4o mini".
   - **Import Library**:

```
import openai
from llama_index.agent.openai import OpenAIAgent
from llama_index.embeddings.openai import OpenAIEmbedding
from llama_index.llms.openai import OpenAI
```

3. **Step 2**

   - **Set API Keys securely**:

```
os.environ["OPENAI_API_KEY"] = os.getenv("OPENAI_API_KEY")
openai.api_key = os.environ.get("OPENAI_API_KEY")
os.environ["LITERAL_API_KEY"] = os.getenv("LITERAL_API_KEY")
```

- **Initialize the Literal AI client**:

```
lai = LiteralClient(api_key=os.environ.get("LITERAL_API_KEY"))
lai.instrument_openai()
```

- **Initialize query engine**:

```
query_engine = query_engine = SubQuestionQueryEngine.
    from_defaults(
        query_engine_tools=individual_query_engine_tools,
        llm=OpenAI(model="gpt-4o-mini"),
)

query_engine_tool = QueryEngineTool(
    query_engine=query_engine,
    metadata=ToolMetadata(
        name="sub_question_query_engine",
        description="useful for when you want to answer queries
            that require analyzing multiple SEC 10-K documents
            for Microsoft",
    ),
)

tools = individual_query_engine_tools + [query_engine_tool]
agent = OpenAIAgent.from_tools(tools, verbose=True)
```

4. **Step 3**

- **Import the library in our chatbot.py file**:

```
import chainlit as cl
```

- **Create the first function**: This function prepares our model, memory object, and llm-chain for user interaction. The chain object is passed to the user session, and a name is specified for the session. This name can be used later on to retrieve a specific chain object when a user sends a query. We want the chat bot to keep last 10 history's messages to limit the resources.

```
message_history = deque(maxlen=10)  # Keep the last 10 messages
@cl.on_chat_start
await cl.Message(
        author="Assistant",
        content=(
            "Hello! I'm an AI financial assistant. How may I help
                you?\n\n"
            "Here are some topics you can ask me about:\n"
```

```python
            "- Budgeting and saving tips\n"
            "- Investment advice\n"
            "- Retirement planning\n"
            "- Understanding credit scores\n"
            "- Managing debt\n"
            "- Tax planning\n"
            "- Insurance options\n"
            "Feel free to ask any questions related to these topics!"
        )
    ).send()
async def set_sources(response, msg):
    elements = []
    label_list = []
    for count, sr in enumerate(response.source_nodes, start=1):
        elements.append(cl.Text(
            name="S" + str(count),
            content=f"{sr.node.text}",
            display="side",
            size="small",
        ))
        label_list.append("S" + str(count))
    msg.elements = elements
    await msg.update()
```

- **Create the second function**: Next, we will define the message handling function with the @cl.on-message decorator. This function is responsible for processing user messages and generating responses:

```python
@cl.on_message
async def main(message: cl.Message):

    # Initialize message history as an empty list if not set
    message_history = cl.user_session.get("message_history", deque(maxlen=10))

    if not isinstance(message_history, list):
        message_history = []
        cl.user_session.set("message_history", message_history)

    # Generate a unique chat_id for this session
    chat_id = str(uuid4())
    payload = {
        "chat_id": chat_id,
        "message": message.content,
        "timestamp": datetime.datetime.now(datetime.timezone.utc).isoformat(),
    }

    # Save the message to Literal AI
    try:
        lai.chat.completions.create(
            model="gpt-3.5-turbo",
            messages=[payload],
        )
        print("Message saved to Literal AI.")
    except Exception as e:
        print(f"Error saving message to Literal AI: {e}")

    msg = cl.Message(content="", author="Assistant")
    user_message = message.content

    # Process the user's query asynchronously
    res = query_engine.query(message.content)

    # Check if 'res' has a 'response' attribute for the full
       response
    if hasattr(res, 'response'):
        # Send the full response in one go
        msg.content = res.response
        message_history.append({"author": "Human", "content":
            user_message})
        message_history.append({"author": "AI", "content": msg.
            content})
        message_history = list(message_history)[-4:]  # Keep the
            last 4 messages
        cl.user_session.set("message_history", message_history)
    else:
        # If res does not have a 'response' attribute, output a
            generic message
        msg.content = "I couldn't process your query. Please try
            again."

    await msg.send()
    if res.source_nodes:
        await set_sources(res, msg)
```

In the code above, we retrieve the previously stored llm-chain object from the user session. This object holds the state and configuration needed to interact with the language model. The llm-chain.acall method is then called with the content of the incoming message. This method sends the message to the LLM, incorporating any necessary context from the conversation history, and awaits the response. Once the response from the LLM is received, it's formatted into a cl.Message object and sent back to the user.

5. **Create the third function**

```python
@cl.on_chat_resume
async def resume():
    try:
        # Ensure lai.chats.list() is awaited if it's an async function
        chat_history = await lai.chats.list() if asyncio.iscoroutinefunction(lai.chats.list) else lai.chats.list()

        if chat_history:
            for message in chat_history:
                # Append the message to local memory and send it to the chat interface
                await cl.Message(content=message['message']).send()
            print("Chat history loaded successfully!")
        else:
            print("No chat history found.")

    except Exception as e:
        print(f"Error retrieving chat history: {e}")
        await cl.Message(content="Sorry, I was unable to load the chat history.").send()

    # Send a message indicating the session has resumed
    await cl.Message(content="Welcome back! How can I assist you today?").send()
```

In the function above, we build it to retrieve the chat history with this chatbot.

Chapter 4

Authentication

In this project, we use Google OAUTH for user to login. There are some benefits when we choose Google OAUTH, but the majors are:

- **User Identification**: We can confirm the validation of users.
- **Security**: Google OAUTH is very secure

1. **Code Highlights**

   - **Import necessary library**:

   ```python
   import os
   ```

   - **Set environment variables for OAuth provider**:

   ```python
   os.environ['OAUTH_GOOGLE_CLIENT_ID'] = os.getenv("
       OAUTH_GOOGLE_CLIENT_ID")
   os.environ['OAUTH_GOOGLE_CLIENT_SECRET'] = os.getenv("
       OAUTH_GOOGLE_CLIENT_SECRET")
   ```

   - **Create "callback" function**: We build this function to make sure that users are valid when they login.

   ```python
   @cl.oauth_callback
   def oauth_callback(
       provider_id: str,
       token: str,
       raw_user_data: dict[str, str],
       default_user: cl.User
   ) -> Optional[cl.User]:
       """Handle Google OAuth callback."""
       print("OAuth callback received from provider:", provider_id)
       print("Token:", token)
       print("Raw user data:", raw_user_data)

       # Check if the provider is Google and process user
           information
       if provider_id == "google":
           user_email = raw_user_data.get("email")
           if user_email:
   ```

```python
            return cl.User(identifier=user_email, metadata={"role": "user"})
    return None
```

Chapter 5

Data processing

In this project, we use finance data to build chatbot database. We use 20 html files that point to 6 Wikipedia web pages to collect data. The reasons behind it are Wikipedia is a well-known website for its high accurate data and easy to approach.

1. **Code Highlight**

   - **Import necessary library**:

```python
from pathlib import Path
from llama_index.readers.file import UnstructuredReader
from llama_index.core.tools import QueryEngineTool, ToolMetadata
from llama_index.core.query_engine import SubQuestionQueryEngine
```

   - **Get the number of items in the data folder**:

```python
def get_document_numbers(data_folder):
    # Convert the data_folder to a Path object
    data_folder_path = Path(data_folder)

    # Check if the data folder exists
    if not data_folder_path.exists():
        raise FileNotFoundError(f"The system cannot find the path
            specified: '{data_folder}'")

    # List all files in the data folder
    files = data_folder_path.iterdir()

    # Filter out only HTML files (not directories)
    html_files = [f for f in files if f.is_file() and f.suffix ==
        '.html']

    # Generate a list of numbers based on the number of HTML
        files
    numbers = list(range(len(html_files)))
    return numbers
```

   - **Initialize UnstructuredReader and load the documents**: We store data in "./data/-Finance"

```
loader = UnstructuredReader()
doc_set = {}
all_docs = []
for number in numbers:
    finance_docs = loader.load_data(
        file=Path(f"./data/Finance/Finance{number}.html"),
            split_documents=False
    )
    for d in finance_docs:
        d.metadata = {"number": number}
    doc_set[number] = finance_docs
    all_docs.extend(finance_docs)
```

- **Initialize vector indices with chunk size**

```
Settings.chunk_size = 512
index_set = {}
for number in numbers:
    storage_dir = Path(f"./storage/finance/{number}")
    storage_dir.mkdir(parents=True, exist_ok=True)

    # Check if index already exists in storage
    if not any(storage_dir.iterdir()):
        # Create and persist index if it doesn't exist
        storage_context = StorageContext.from_defaults()
        cur_index = VectorStoreIndex.from_documents(
            doc_set[number],
            storage_context=storage_context,
        )
        index_set[number] = cur_index
        storage_context.persist(persist_dir=storage_dir)
    else:
        # Load existing index from storage
        storage_context = StorageContext.from_defaults(
            persist_dir=storage_dir)
        index_set[number] = load_index_from_storage(
            storage_context)
```

- **Create tools for each number index if not already created**

```
individual_query_engine_tools = [
        QueryEngineTool(
            query_engine=index_set[number].as_query_engine(),
            metadata=ToolMetadata(
                name=f"vector_index_{number}",
                description=f"useful for when you want to answer
                    queries about the {number} for finance
                    documents",
            ),
        )
        for number in numbers
    ]
```

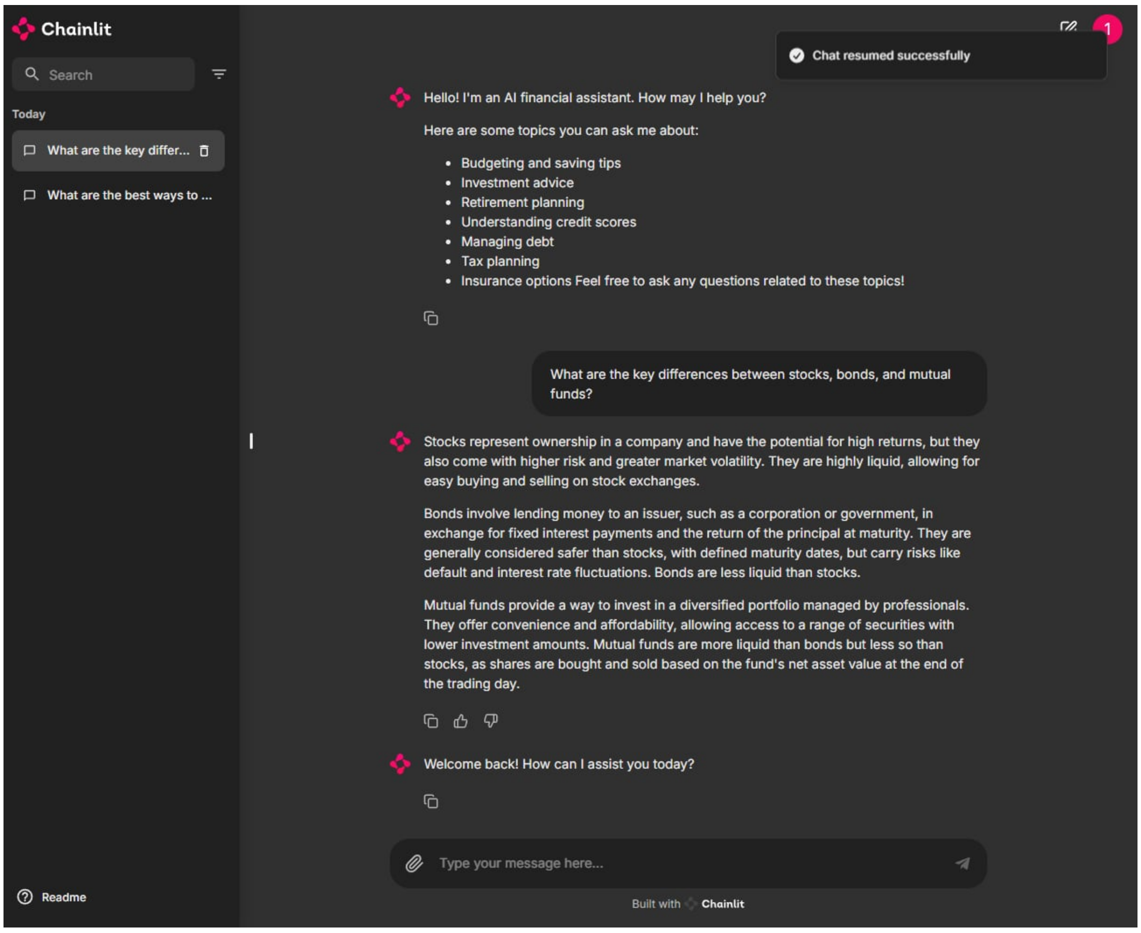
Chapter 6

Testing and Evaluation

In this chapter, we will test and evaluate our chat bot.

6.1 Testing the Chatbot

6.1.1 Functional Testing


Chainlit
Search
Today
What are the key differ...
What are the best ways to ...
Readme
Chat resumed successfully
Type your message here...
Built with Chainlit

Hello! I'm an AI financial assistant. How may I help you?

Here are some topics you can ask me about:

- Budgeting and saving tips
- Investment advice
- Retirement planning
- Understanding credit scores
- Managing debt
- Tax planning
- Insurance options Feel free to ask any questions related to these topics!

What are the key differences between stocks, bonds, and mutual funds?

Stocks represent ownership in a company and have the potential for high returns, but they also come with higher risk and greater market volatility. They are highly liquid, allowing for easy buying and selling on stock exchanges.

Bonds involve lending money to an issuer, such as a corporation or government, in exchange for fixed interest payments and the return of the principal at maturity. They are generally considered safer than stocks, with defined maturity dates, but carry risks like default and interest rate fluctuations. Bonds are less liquid than stocks.

Mutual funds provide a way to invest in a diversified portfolio managed by professionals. They offer convenience and affordability, allowing access to a range of securities with lower investment amounts. Mutual funds are more liquid than bonds but less so than stocks, as shares are bought and sold based on the fund's net asset value at the end of the trading day.

Welcome back! How can I assist you today?

- Function for Question and Answer with user works successfully. It can answer question about finance such as explain some definitions, give some information about finance history, ...
- We have recreated Chainlit UI successfully. The UI is very smooth and easy for user to interact with.

- We also create a chatbot history to store the last 10 messages with user. In our point of view, we consider 10 is enough for user because it balances between customer's needs and our resources. It also recognizes old users with the chatbot history.
- We also successfully deploy our chatbot locally.

6.2 Performance Evaluation

- Our chatbot needs approximately 1 seconds to answers easy questions such as explaining definitions, ... When it approachs some more difficult questions, it needs more time base on the complexity of the questions, the average is about 3 to 6 seconds.
- With easy questions, it can answer very accurate, but when comes to some difficult ones, the accuracy is lower or it may gives wrong information.

Chapter 7

Conclusion and Future Work

In this chapter, we will discuss about what we have achieved during our project, what are the limitations that we faced and the improvements that our chatbot needed.

7.1 Summary of Key Achievements

- **Task list**: In our project, we have fulfilled all of the objectives from our supervisor, which are " A complete chatbot that can do QAs for a topic", "Using Chainlit to create UI for your chatbot", "Using Chainlit to create a memory for your chatbot.", "Deploy your app locally"
- **Enhancement**: We have some minor improvements such as upgrading user-interface, improve the performance of the chatbot with some changes in the code.

7.2 Limitations

- **Performance**: Our data training for chatbot base one "OpenAI 4o mini", the model which have some limitations with performance due to the price. We also didn't have the funds to acquire more powerful models and keep the data training continuously
- **User interface**: Our chatbot base on Chainlit UI, which is free, so the completeness and the complexity does not reach our goals.
- **Data**: We also need to collect and add data manually so it cost a huge amount of time.

7.3 Future Enhancements

- If we have more money to spends, it is easy for us to upgrade the performance and also the UI. We can build a tool that automatically collect and add data for us.

Chapter 8

References

- "https://docs.llamaindex.ai/en/stable/understanding/puttingitalltogether/chatbots/buildingachatbot/"