Vietnamese – German University
DEPARTMENT OF COMPUTER SCIENCE

Frankfurt University of Applied Sciences
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

Building a Custom Chatbot using LlamaIndex and Custom Data

Authors:	Tu Trung Tin Nguyen Nho Minh Tri
Matriculation Numbers:	10421106 10421061
Professor:	Dr. Dinh Quang Vinh

COMPULSORY ELECTIVE 2

SUBMITTED IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE COMPULSORY ELECTIVE 2 IN STUDY PROGRAM COMPUTER SCIENCE, VIETNAMESE-GERMAN UNIVERSITY, 2023

Binh Duong, Vietnam

Contents

Chapter 1

Introduction

In today's world, the complexity, and interdependence of economic systems underscore the importance of accessible knowledge in economics. Understanding basic economic, principles is crucial not only for policymakers and business leaders but also for the general public. However, traditional methods of learning economics, often laden with technical jargon, and complex graphs, can create barriers for many individuals.

1.1 Motivation

The motivation behind this project stems from a desire to bridge the gap between economics and its audience by leveraging advancements in artificial intelligence. Thomas Sowell's Basic Economics provides an excellent foundation for this initiative, with its clear, jargon-free explanations of economic concepts. By combining this invaluable resource with the powerful capabilities of LlamaIndex, we aim to create a chatbot that democratizes economic knowledge, making it accessible to anyone with a question about the subject..

1.2 Objective

The primary objective of this project is to design, and implement a conversational chatbot capable of answering queries related to economic principles, as presented in Basic Economics. The chatbot will provide users with detailed, accurate, and contextually relevant responses by leveraging LlamaIndex to structure, and search through the book's content effectively. This tool aims to promote economic literacy, and serve as a valuable learning aid for students, educators, and enthusiasts alike.

Chapter 2

Preparation

In this part, we will set up, and install necessary components for the chatbot project.

2.1 Installation

If you have not already cloned this project to your local machine, do so now:

```
 git clone --recursive https://github.com/empty-nn/llama-chainlit-chatbot
-.git

```

2.2 Library

2.2.1 Standard Python Library

For this project, we use various libraries, and packages for file handling, document parsing, natural language processing, AI integration, and web interface development. Here's an overview of the libraries:

- os
- nest_asyncio
- pathlib
- uuid
- collections
- datetime
- typing

2.2.2 Third-party Library

Furthermore, our project also use some third-party library for initiating chatbot functionality. Here's an overview of the libraries, and their functions:

1. chainlit:
   - An open-source Python library that makes it easier to build production-ready chatbot applications. It focuses on managing user sessions, and their events, such as message exchanges, and user queries.

2. llama_index:
   - An open-source data orchestration framework for developing large language model (LLM) applications.

3. nltk:
   - A collection of libraries, and programs for symbolic, and statistical natural language processing (NLP) for English written in Python.

2.3 Data Preparation

In this project, we are using the PDF book named Basic Economics by Thomas Sowell as the core knowledge base.The purpose of this application is to automate the process of extracting, and storing their content, as well as to arrange the saved information into an arranged folder.

Chapter 3

How to Build a Chatbot

LlamaIndex serves as a bridge between your data, and Large Language Models (LLMs), providing a toolkit that enables you to establish a query interface around your data for a variety of tasks, such as question-answering, and summarization.

In this section, we outline the process of building a context-augmented chatbot using a Data Agent, which can intelligently execute tasks over your data. The chatbot is powered by LLMs, and uses LlamaIndex tools to answer queries about data. The following steps illustrate the development of a "Economic Chatbot" using Basic Economics book written by Thomas Sowell.

3.1 Before Run the Code

You're working on a project involving text analysis, processing, or interpretation. nltk is a must-have library. Here is the way you can install it.

```
pip install nltk

```

Then you can verify the installation by:

```
python

```

After that, you need to download Additional NLTK Resources. Many features in nltk require additional data files. To download these resources:

```
import nltk
nltk.download()

```

Now, you can continue with your work.

3.2 Preparation

3.2.1 Import Required Libraries

- **os**: Retrieves the variables form the the system environment variables.
- **openai**: Configures OpenAI's Python SDK(openai) to use the retrieved API key
- **nest_asyncio**: Prevents event loop errors when running asynchronous code.

```
import os
import openai
import nest_asyncio

```

3.2.2 Configure system environment variables

First, the code configures API keys, and initializes libraries required for working with OpenAI's language models, and Google OAuth. These keys, and ids are extracted from the .env file.

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

    openai.api_key = os.getenv("OPENAI_API_KEY")

    nest_asyncio.apply()

```

The code phrase *nest_ asyncio.appy()* applies the patch to make asynchronous code run smoothly in environments like Jupyter Notebooks or applications requiring nested loops.

3.3 Ingest Data & Set up Vector Indices

The code provided here takes a PDF book, extracts its content, creates structured document entries, and persists these entries in a vector store. This allows for fast, and efficient retrieval of information during interactive user queries.

3.3.1 Import Required Libraries

- **PDFMiner**: Used to extract text from the PDF book.
- **llama_index.core**: Provides tools like `StorageContext`, `VectorStoreIndex`, and `Document` to facilitate indexing, and storage.
- **Pathlib**: Used to handle file paths.

```
    from pdfminer.high_level import extract_text
    from llama_index.core import Settings, StorageContext, VectorStoreIndex
    from llama_index.core import Document
    from pathlib import Path

```

3.3.2 Extract Text from PDF

A function named *read_ pdf()* is defined to extract text from the given PDF file using **PDFMiner**. This step converts the raw content of the PDF into a readable text format.

```
    # Function to extract text from PDFs using PDFMiner
    def read_pdf(file_path):
        try:
            text = extract_text(file_path)
            return text
        except Exception as e:
            print(f"Error reading {file_path}: {e}")
            return ""

```

The file path for the book is specified as `book_path`, and the *read_pdf()* function extracts the content of the PDF:

```
    book_path = Path("./data/economics.pdf")
    pdf_content = read_pdf(book_path)

```

3.3.3 Create Document Entries

The extracted content is then wrapped in a dictionary with **metadata** (title of the book). This content is used to create `Document` objects, which are suitable for indexing.

```
    # Create document entry and add metadata
    book_docs = [{"content": pdf_content, "metadata": {"title": "Basic
  Economics"}}]

    # Convert documents into the required format for indexing
    documents = [Document(text=doc["content"], metadata=doc["metadata"]) for
    doc in book_docs]

```

3.3.4 Create Storage Context, and Index:

A **storage_context** is then created to manage how the data is stored. The extracted content is indexed using `VectorStoreIndex` to facilitate fast, and effective querying.

```
    Settings.chunk_size = 512
    # Create a storage directory for the book index
    storage_dir = Path("./storage/book")
    storage_dir.mkdir(parents=True, exist_ok=True)

    # Create a storage context and vector store index
    storage_context = StorageContext.from_defaults()
    book_index = VectorStoreIndex.from_documents(documents, storage_context=
  storage_context)

    storage_context.persist(persist_dir=storage_dir)

```

The storage context, along with the created index, is persisted to the disk for later use. This ensures that the work done during indexing doesn't need to be repeated every time the application is run.

3.4 Set up the Chatbot Agent

3.4.1 Import Required Libraries

- **llama_index.core.tools**: Provides tools like QueryEngineTool , ToolMetadata, and Document to facilitate semantic querying of data, and supports integration into custom workflows facilitate indexing, and storage.
- **llama_index.core**: Provides tools like Settings, load_index_from_storage, and Document to customizes indexing behavior, and ensures persistence across sessions.

- **llama_index.llms.openai**: Provides the language model backend for processing queries, and generating responses.
- **llama_index.agent.openai**: Builds conversational agents or assistants that require complex reasoning, and decision-making.

```python
from llama_index.core import StorageContext
from llama_index.core.tools import QueryEngineTool, ToolMetadata
from llama_index.core import Settings, load_index_from_storage
from llama_index.llms.openai import OpenAI
from llama_index.agent.openai import OpenAIAgent

```

3.4.2 Load Vector Indices

The code retrieves a previously created index for a book from a storage directory. This enables querying or updating the index without needing to recreate it.

```python
# Make sure this matches the directory you used for storing the book index
storage_dir = "./storage/book"

storage_context = StorageContext.from_defaults(
    persist_dir=storage_dir
)

# Load the previously saved index for the book using load_index_from_storage
book_index = load_index_from_storage(storage_context)

```

3.4.3 Set up the Chatbot Agent

The **QueryEngineTool** is created for querying a book index, and integrate it with an **OpenAIAgent**. The tool abstracts query operations, while metadata provides a clear description of its purpose. The agent uses OpenAI's models to interpret user inputs, and decide when to utilize the tool, enabling natural language interactions.

```python
#Create a Query Engine Tool for the Book Index
query_engine_tool = QueryEngineTool(
    query_engine=book_index.as_query_engine(),
    metadata=ToolMetadata(
        name="book_index",
        description="Index for the book - useful for answering questions about the basic economics"
    )
)

#Create an OpenAIAgent using the Book Query Tool
agent = OpenAIAgent.from_tools([query_engine_tool], verbose=True)


```

3.4.4 Test Agent

```python
# Example Usage: Querying the Content of the Book
response = agent.chat("Can you summarize the main topics covered in this book?")
print(response)

```

Output:

```
Added user message to memory: Can you summarize the main topics covered in this book?
=== Calling Function ===
Calling function: book_index with args: {"input":"main topics"}
Got output: The main topics include the nature of markets and economic transactions,
myths about market pricing, the implications of cultural isolation on economics,
the challenges of enforcing anti-dumping laws, the benefits of free trade for
small economies, the reasoning behind costly safety measures, price variations
for identical products, differences between mercantilist and classical economic
theories, the role of branding in consumer products, and historical perspectives
on the relationship between prices and production costs.
========================

The main topics covered in the book include:

1. The nature of markets and economic transactions.
2. Myths about market pricing.
3. Implications of cultural isolation on economics.
4. Challenges of enforcing anti-dumping laws.
5. Benefits of free trade for small economies.
6. Reasoning behind costly safety measures.
7. Price variations for identical products.
8. Differences between mercantilist and classical economic theories.
9. The role of branding in consumer products.
10. Historical perspectives on the relationship between prices and production costs.
```

Chapter 4

User Interface with Chainlit

Chainlit is an open-source Python package to build production ready Conversational AI. Key features:

- **Real-time Communication**: Supports real-time messaging for smooth interaction.
- **Customization**: Allows developers to tailor the UI using pre-built or custom components.
- **Seamless Integration**: Works well with frameworks like LlamaIndex for backend, and frontend communication.
- **Developer Friendly**: Hot reloading, and a robust environment for fast iteration.

4.1 Installation

Chainlit requires python>=3.8. You can install Chainlit it via pip as follows:

```
pip install chainlit

```

4.2 Basic setup Chainlit

```python
import chainlit as cl

@cl.on_message
async def main(message: cl.Message):
    # Your custom logic goes here...

    # Send a response back to the user
    await cl.Message(
        content=f"Received: {message.content}",
    ).send()

```

To start Chainlit app, open a terminal, and run following command:

```
chainlit run app2.py -w

```

4.3 Basic UI for chatbot

4.3.1 Setting up the Agent

```python
from llama_index.core import Settings, StorageContext,
load_index_from_storage
from llama_index.core.tools import QueryEngineTool, ToolMetadata
from llama_index.llms.openai import OpenAI
from llama_index.agent.openai import OpenAIAgent

storage_dir = "./storage/book"
storage_context = StorageContext.from_defaults(persist_dir=storage_dir)
book_index = load_index_from_storage(storage_context)

Settings.llm = OpenAI(model="gpt-4o-mini", temperature=0.2)

query_engine_tool = QueryEngineTool(
    query_engine=book_index.as_query_engine(),
    metadata=ToolMetadata(
        name="book_index",
        description="Index for the book - useful for answering questions
about basic economics"
    )
)

agent = OpenAIAgent.from_tools([query_engine_tool], verbose=True)

```

This agent is the core of our chatbot, capable of understanding questions related to the indexed book content, and providing insightful answers.

4.3.2 Initialize the chatbot

```python
def initialize_chatbot_for_years(memory):
agent = OpenAIAgent.from_tools(
    tools=[query_engine_tool],
    memory=memory
)

return agent

```

The *initialize_chatbot_for_years* function creates, and configures a chatbot agent by combining tools (such as query engines) with a memory buffer.

4.3.3 Integrating the Agent with Chainlit

Once the agent is set up, we can integrate it into Chainlit to build a user-friendly UI. Chainlit provides tools for real-time interaction, which helps us leverage the capabilities of the OpenAIAgent effectively. Here is how to set up the basic chatbot interface:

```python
import chainlit as cl

@cl.on_chat_start
async def initialize_chat_session():
    await cl.Message(
        author="assistant",
        content="Hello! I'm an AI assistant. How may I help you?"
    ).send()

@cl.on_message
async def handle_user_message(message: cl.Message):
    # Use the agent to get a response for the user message
    response = agent.chat(message.content)

    # Send response back to the user
    await cl.Message(
        content=response
    ).send()

```

Now, you have a UI for chatbot but it can not save the history. It can only handle user messages in a chatbot session.

4.4 Advanced

4.4.1 Chat History

Chat history allows users to search, and browse their past conversations. To enable chat history, you need to enable:

- **Data persistence**: You can follow this guide **HERE** to enable Data Persistence
- **Authentication**: You can use Password Auth, Oauth or Header Auth. However, in our project, we would prefer using Oauth. In particular, Google Oauth is the one we choose for our authentication system in our web application. You can follow this guide **HERE** to enable Google Oauth or go to **HERE** to understand how to enable other authentication system.

Then, you need to process your callback function to handle the Google OAuth process in an application, where the function is triggered after a user successfully authenticates with Google.

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

```

The function *oauth_ callback* is responsible for processing user authentication data received from Google after a successful OAuth login. It validates the data, and returns a user object that the application can use to manage the user's session, and access permissions. Not only that, the chainlit platform registers the function as the OAuth callback handler for the application.

4.4.2 Memory for Agent

First, we need to edit the function handling the incoming user messages in the chatbot framework. In this case, we will set up the memory in the *handle_user_message*.

```python
@cl.on_message
async def handle_user_message(message: cl.Message):
    # Retrieve the chat store path from the user session
    history_path = cl.user_session.get("history_path")
    memory = cl.user_session.get("memory")

    # Initialize context to get thread_id for unique identification
    context = get_context()
    thread_id = context.session.thread_id

    # Determine if this is a new session or an existing one
    if history_path is None:
        # Create a new path for the chat history based on thread_id
        history_path = Path(f"./history/{thread_id}.json")
        # Ensure the directory exists
        history_path.parent.mkdir(parents=True, exist_ok=True)
        # Initialize a new chat store
        chat_store = SimpleChatStore()
        # Save history_path in the session for future use
        cl.user_session.set("history_path", str(history_path))
        memory = ChatMemoryBuffer.from_defaults(
            token_limit=3000,
            chat_store=chat_store,
            chat_store_key= thread_id  # Using username as a key to
store conversation history
        )

    cl.user_session.set("memory", memory)

    # Extract the content of the user's message
    message_content = message.content

    # Initialize an agent (assuming this uses chat_store)
    agent = initialize_chatbot_for_years(memory)

    # Generate a response from the assistant
    response = str(agent.chat(message_content))

    # Persist the updated chat store to the history file
    memory.chat_store.persist(str(history_path))

    # Send the assistant's response back to the user
    await cl.Message(content=response).send()

```

In this function, it creates, handles new, and existing sessions by storing chat histories on disk.

In the next step, in regard to adding chat history, resuming a chat session is an essential feature for maintaining continuity in user interactions with the chatbot. This section outlines the steps, and implementation plans for enabling chat resumption, which includes retrieving stored conversation history, and reintegrating it into the chatbot's context during subsequent sessions.

```
@cl.on_chat_resume
async def resume_chat_session(thread: ThreadDict):
    history_path = cl.user_session.get("history_path")
    history_path = Path(history_path)  # Convert to Path object if it was stored as a string
    chat_store = SimpleChatStore.from_persist_path(str(history_path))

    context = get_context()
    thread_id = context.session.thread_id

    memory = ChatMemoryBuffer.from_defaults(
        token_limit=3000,
        chat_store=chat_store,
        chat_store_key= thread_id  # Using username as a key to store conversation history
    )
    cl.user_session.set("memory", memory)

```

Chapter 5

Testing & Evaluation

5.1 Objectives of Testing

This project uses data from Thomas Sowell's Basic Economics PDF book. We seek to analyze numerous essential capabilities, including chat history, normal login, and alternative login ways using Google authentication.

Only logged-in users can access the chat history function on the chatbot's webpage.

This testing process guarantees the following:

- The chatbot provides users with clear, and relevant updates by retrieving, and summarizing recent news.
- Users can access past chat logs while maintaining privacy through distinct login processes.

These tests will check the chatbot's technical capabilities, and usability to provide a secure, and user-friendly experience.

5.2 Test Plan

- Function testing: We will test the chatbot's accuracy, support Google login, and ensure chat logs are kept, and retrievable on disk, and on Literalai threads.
- Performance testing: We will give a prompt, and measure its response to ensure that it will run under 5 minutes. If you give it a complex question, it will take more time to give a response. Due to performance bottleneck in large indices, our chatbot will face latency issues when querying large datasets due to a single query engine.
- Accuracy Testing: We will compare the chatbotś generated summaries, and responses to the actual content provided by our PDF book to evaluate the precision, and reliability of information retrieval.

5.3 Testing & Results

Based on our demo video we provide, user can login with google account.

The result of the chatbot is extracting, analyzing from the given data. It gives out the pleasing output which is understandable, appropriate to what we expect.

- **"Explain the principles of supply and demand"**: is the return of the principles of supply and demand principle and the explaination about how the principles interacted with the relationship between the quantity of goods, services, and the desire of consumers.

- **"Explain the concept of productivity"**: is the return of the concept of productivity and some key points about it.

Chapter 6

Conclusion

In the rapidly developing field of artificial intelligence, there is enormous potential for developing conversational agents that are both clever, and data-specific by utilizing tools like as LlamaIndex to create unique chatbots. Developers can create responsive solutions that are closely matched to certain use cases by combining LlamaIndex with a customized dataset. This strategy enables businesses to optimize the value of their distinct datasets, resulting in more individualized user experiences, and increased user engagement in general.

Although issues like scalability, model fine-tuning, and data preprocessing still exist, LlamaIndex's flexibility guarantees that these may be methodically resolved. Furthermore, such chatbots can continuously improve in accuracy, and efficacy by implementing systematic techniques, and iterating depending on input. In conclusion, custom chatbot development using LlamaIndex demonstrates the transformative impact of AI in reshaping how we interact with technology, paving the way for more human-centric, and context-aware applications.