Contents

1. Introduction

In this project, I aim to develop a chatbot that can handle question-and-answer (Q&A) tasks based on a specific dataset. The chatbot leverages the capabilities of the LlamaIndex library alongside Chainlit to provide a user-friendly interface and memory features, allowing it to engage in conversations with users.

This whole project is structured into several key steps: reproducing existing code [1], preparing the dataset[2], creating the chatbot, and finally documenting the entire process.

The initial phase involved understanding the provided code[1] and configuring it in my IDE. Given the nature of the chatbot, I needed to ensure that the code could effectively read, process, and respond to user queries based on the input data. By diving deep into the implementation details and solving problems related to environment setup and API access, I gained valuable insights into how chatbots function and how to manage data effectively.

2. Initial Setup and Code Explanation

2.1 Setting Up the Environment

The first step in this project was to set up my coding environment. I chose to use PyCharm as my integrated development environment (IDE) instead of Jupyter Notebook. This decision was driven by several factors:

- **Familiarity**: I am more comfortable using PyCharm, which has robust features for handling Python code, debugging, and managing project files.
- **Import Issues**: During initial attempts to run the code in the Jupyter Notebook, I encountered several import errors that hindered my progress e.g: "module Python magic not found" even though this module was already imported. PyCharm provided a more stable environment for resolving these kind of issues.

2.2 Code Reproduction and Import Management

The next task involved copying the code from the provided source from our google classroom [3]

After copying the code, I encountered numerous import errors. This was largely due to dependencies not being correctly installed or recognized in my environment. To resolve this, I made sure that all required libraries were installed using pip:

pip install chainlit, llama_index, openai, nest_asyncio, python-dotenv were additionally needed.

Once the necessary libraries were installed, I moved forward with testing the code. Initially, I used Uber 10k SEC filings as my dataset to check if the chatbot was functioning correctly. This data was also provided in the previously mentioned link However, I quickly realized that my program was not operational because it required an OpenAI API key to make requests.

2.3 Obtaining the OpenAI API Key

Recognizing the need for an API key, I conducted some research on how to obtain one. It is actually free to obtain a key but Open-Ai charges the user for every use of the Key in a program. After evaluating the costs and benefits, I decided to load 5€ into my account. This decision was necessary to enable the

chatbot to access OpenAI's language models, which are crucial for generating responses to user queries. I realised that using an Open-Ai API is really cheap. In average it costs me less than 1 cent per request (question), however, one can imagine how expensive it can become, when multiple clients are using the same API key.[6]

2.4 Understanding the Code Functionality

```python
import os
import chainlit as cl
from llama_index.agent.openai import OpenAIAgent
from llama_index.core.tools import QueryEngineTool, ToolMetadata
from llama_index.llms.openai import OpenAI
from llama_index.core.query_engine import SubQuestionQueryEngine
from llama_index.readers.file import UnstructuredReader
from llama_index.core import VectorStoreIndex, StorageContext, load_index_from_storage, Settings
import openai
import nest_asyncio
from pathlib import Path
from dotenv import load_dotenv
```

After successfully obtaining the API key and fixing the initial errors, I took the time to understand the functionality of the code. Below is a detailed explanation of the code snippet provided:

1. import os
   - Purpose: This module provides a way to interact with the operating system, allowing access to environment variables and handling file paths.
   - Why Needed: os is used here to retrieve keys like the OPENAI_API_KEY or the Login data of code, making it safer and more flexible.
2. import chainlit as cl
   - Purpose: chainlit is a framework for creating interactive chat UIs and managing user sessions in chat applications.[4]
   - Why Needed: In this project, chainlit is used to build the chatbot's UI and manage messages between the user and the chatbot. After executing the program, a local server will be created where the user can login and communicate with the agent in a beautiful environment.
3. from llama_index.agent.openai import OpenAIAgent
   - Purpose: The OpenAIAgent class provides a way to interact with OpenAI's models, abstracting some of API calls.
   - Why Needed: This is the core class that allows the chatbot to communicate with the OpenAI API, enabling it to perform natural language understanding and response generation.
4. from llama_index.core.tools import QueryEngineTool, ToolMetadata

- Purpose: QueryEngineTool and ToolMetadata are classes for defining and handling various querying tools that the chatbot can use to retrieve information from its dataset.
- Why Needed: These tools allow the chatbot to query different datasets or indexes more flexibly. ToolMetadata provides information for each tool, helping organize responses and metadata for each query. It is like a performance booster.

5. from llama_index.llms.openai import OpenAI
   - Purpose: The OpenAI class provides a direct interface to OpenAI's language models (In my program I use GPT-3.5-turbo).
   - Why Needed: This is used as the language model for generating responses. It provides a way to interact with OpenAI's models, allowing easy customization of model parameters and configurations.
6. from llama_index.core.query_engine import SubQuestionQueryEngine
   - Purpose: The SubQuestionQueryEngine class is designed to handle more complex queries that may involve multiple sub-queries or different parts of a dataset.
   - Why Needed: This enables the chatbot to break down complex queries into manageable parts, searching across various documents or indexes to generate a comprehensive response. It's useful for datasets that are split across different categories (like floors in this project).
   -
7. from llama_index.readers.file import UnstructuredReader
   - Purpose: UnstructuredReader is a utility class that reads and processes unstructured text files.
   - Why Needed: This reader is used to load text files (like the "Heavenly Tower" floors data) into the program so that the chatbot can use them as a source of knowledge for answering questions. It enables the chatbot to read raw text files without needing specific data formatting. In my project I only used .txt formatted files.

8. from llama_index.core import VectorStoreIndex, StorageContext, load_index_from_storage, Settings
   - Purpose: These classes and functions manage the chatbot's vector indexes and storage.
     - VectorStoreIndex: Creates a vector-based searchable index of the text data.
     - StorageContext: Manages the storage settings and configurations for the index.

  - load_index_from_storage: Loads pre-saved indexes from disk.
  - Settings: Allows customization of indexing parameters, like chunk_size.

- Why Needed: These tools are essential for indexing the documents so the chatbot can perform fast and relevant searches. The code uses this functionality to create and load indices for each floor, making querying efficient and quick.

9. import openai

- Purpose: This library provides access to OpenAI's API functions, allowing calls to OpenAI models for generating responses.
- Why Needed: This import is needed for directly setting the API key and making requests to OpenAI's language models.

10. from pathlib import Path

- Purpose: The Path class from the pathlib library provides an approach to handling file paths.
- Why Needed: This is used here to manage file paths for loading data. It allows the program to construct paths for each floor's text file in a clear way.

11. from dotenv import load_dotenv

- Purpose: load_dotenv loads environment variables from a .env file into the Python environment.
- Why Needed: This is important for securely storing sensitive information, such as the OPENAI_API_KEY. Using load_dotenv, I can easily keep sensitive data separate from the main code and out of version control.

```
14  # Load environment variables from .env file
15  load_dotenv()
16  openai.api_key = os.getenv("OPENAI_API_KEY")
17
18  # Apply nest_asyncio to allow nested event loops
19  nest_asyncio.apply()
```

2.4.1 Environment Variable Management

The code begins by loading environment variables from a .env file using the load_dotenv function. This file contains sensitive information such as API keys and credentials that should not be hard-coded in the script for security reasons: After loading the environment variables, I set the OpenAI API key in the environment. This setup is essential for authenticating requests to the OpenAI services.

2.4.2 Document Loading and Indexing

Next, I configured the document loading process. I used a different python script to scrape all the important data from a the website to store it locally (more in chapter 3.0). The use of a loop allows the program to load documents for all 50 floors efficiently.

```
# Document loading and indexing
floors = list(range(1, 51))  # List of floors from 1 to 50
loader = UnstructuredReader()
doc_set = {}

# Load documents for each floor
for floor in floors:
    # Load data for the current floor
    levels = loader.load_data(file=Path(f"./heavenly_tower_floors/floor_{floor}.txt"), split_documents=False)

    # Add metadata for each document
    for d in levels:
        d.metadata = {"floor": floor}  # Add floor metadata

    # Store documents in the dictionary
    doc_set[floor] = levels

# Initialize simple vector indices
Settings.chunk_size = 512
index_set = {}
```

- **Document Structure**: Each floor is represented by a text file, which is generated using a different python script.
- **Metadata Addition**: For each document, metadata is added to specify which floor it corresponds to. This metadata will be helpful later for efficiency when querying.

2.4.3 Creating Vector Store Indices

The next part of the code initializes vector store indices for each floor. This is critical for enabling the chatbot to retrieve information quickly based on user queries:

```
# Initialize simple vector indices
Settings.chunk_size = 512
index_set = {}

# Create vector store indices for each floor
for floor in floors:  # Use floors instead of years
    storage_context = StorageContext.from_defaults()
    cur_index = VectorStoreIndex.from_documents(doc_set[floor], storage_context=storage_context)
    index_set[floor] = cur_index
    storage_context.persist(persist_dir=f"./storage/{floor}")
```

- **Chunk Size**: The chunk size for indexing is set to 512, which defines how the documents will be divided into smaller parts for processing.
- **Index Creation**: For each floor, a vector store index is created and stored on disk. This allows for efficient querying later on.

2.4.4 Loading Indices from Disk

After creating the indices, I also included functionality to load the indices from the disk for quick access:

```
# Load indices from disk
for floor in floors:
    storage_context = StorageContext.from_defaults(persist_dir=f"./storage/{floor}")
    cur_index = load_index_from_storage(storage_context)
    index_set[floor] = cur_index
```

This ensures that the indices can be reused without needing to rebuild them every time the program runs, which improves performance.

2.4.5 Query Engine Tools Creation

```
# Create query engine tools
individual_query_engine_tools = [
    QueryEngineTool(
        query_engine=index_set[floor].as_query_engine(),
        metadata=ToolMetadata(
            name=f"vector_index_{floor}",
            description=f"useful for when you want to answer queries about a single floor",
        ),
    )
]

# Create a sub-question query engine
query_engine = SubQuestionQueryEngine.from_defaults(
    query_engine_tools=individual_query_engine_tools,
    llm=OpenAI(model="gpt-3.5-turbo"),
)

# Query engine tool for complex queries
query_engine_tool = QueryEngineTool(
    query_engine=query_engine,
    metadata=ToolMetadata(
        name="sub_question_query_engine",
        description=f"useful for when you want to answer queries about multiple floors",
    ),
)
```

To handle user queries effectively, I created query engine tools for each floor and a sub-question query engine for complex queries:

- **Query Engine Tool**: Each floor's index is wrapped in a QueryEngineTool, which enables structured querying of the data. This allows retrieval of simple data that is conducted from a maximum of one floor.
- **Sub-Question Engine**: The sub-question engine combines the individual tools, allowing the chatbot to analyze queries that span multiple floors. E.g: give me the names of the floors where the enemy “doge” appears

2.5 Message Handling and User Interaction

Finally, the code defines how the chatbot handles incoming messages from users:

```python
# Combine tools for the agent
tools = individual_query_engine_tools + [query_engine_tool]
agent = OpenAIAgent.from_tools(tools)

# Message handling for incoming user messages
@cl.on_message
async def main(message: str):
    # Retrieve and manage conversation history
    conversation_history = cl.user_session.get("conversation_history") or []
    user_message = message.content
    conversation_history.append({"user": user_message})

    # Generate a response from the agent
    response = agent.chat(user_message)
    conversation_history.append({"bot": response})

    # Update session with new conversation history
    cl.user_session.set("conversation_history", conversation_history)

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

```

- **Session Management**: This part of the code retrieves and updates the conversation history, allowing the chatbot to remember past interactions.
- **Response Generation**: Upon receiving a message, the chatbot generates a response based on the user input and appends both to the conversation history for future reference.
- **Use of "Literal Ai"**: For activating data persistence I used "Literal API" [5]. The API Key is provided in the environmental variables and is accessed by chainlit.

2.6 Authentication

Additionally, I included an authentication so the user needs to log in to an account before accessing the chatbot. I stored the username and the password, aswell as all needed API keys in the environmental variables, so that they are not visible in the code. For testing purposed I will write down the account data here: ADMIN_USERNAME=admin ADMIN_PASSWORD=admin.

```python
# Authentication callback for user login
@cl.password_auth_callback
def auth_callback(username: str, password: str):
    if (username, password) == (os.getenv("ADMIN_USERNAME"), os.getenv("ADMIN_PASSWORD")):
        return cl.User(
            identifier="admin",
            metadata={"role": "admin", "provider": "credentials"}
        )
    else:
        print(os.getenv("ADMIN_USERNAME"))  # Debugging line
        return None
```

This block of code provides a function that returns a successful login when the username and the password equals the credentials that are stored in the environmental variables. Otherwise the function returns nothing and we can not login. To prevent a brute force attack one could limit the amount of times to call the function. For example one could implement that it is not possible to login anymore after 5 tries.

3.0 Preparing the dataset

To prepare my dataset for this project, I started by experimenting with some initial data sources to see if the setup I envisioned would work. My first choice was to pull data that was provided (Uber 10k SEC filings), primarily for testing purposes. I wanted to see if I could load data into my chatbot effectively. After getting it up and running, I successfully pulled and processed data from these filings, which included various details on corporate operations, revenue, and other financial data.

Step 1: Experimenting with SEC Filings and Nike Reports

After confirming that my code could handle large datasets, I looked for data with a bit more structure and a familiar name. This led me to try using Nike's annual reports as an experimental dataset. I could see that the chatbot was able to pull relevant data from Nike's filings, including details like "total revenue," "operating expenses," and even product categories. For example, it could answer questions about Nike's revenue trends or its growth strategy in recent years. Although it was interesting to see the bot handle this kind of corporate data, I quickly realized that the content wasn't engaging or practical for me. The chatbot could respond to financial inquiries, but it wasn't helping me in a meaningful way. This sparked the idea to create a chatbot that would serve a specific personal interest of mine: The Battle Cats game, specifically the challenging Heavenly Tower levels. I used to play this game when I was a kid.

Step 2: Selecting Battle Cats' Heavenly Tower as the Topic

The Battle Cats is a popular mobile game where players create a team of cats to defend their base against various enemies. There are different stages in the game, and one of the hardest set of stages is the Heavenly Tower. The Heavenly Tower

is a special set of 50 challenging levels, each floor presenting unique enemies, difficulties, and strategies. Beating these levels without proper guidance is close to impossible, given the game's complexity and the fact that strategies often vary significantly between floors.

For me, Heavenly Tower had always been a tricky part of the game. I would spend a lot of time researching strategies across different websites, forums, and videos. This was time-consuming and sometimes frustrating, as I had to gather information from various sources and then piece it together. So, the idea of having a chatbot that could provide strategic insights for each floor of the Heavenly Tower was incredibly appealing.

Step 3: Gathering and Preparing the Data with Python

To create this chatbot, I needed detailed data on each floor of the Heavenly Tower. Luckily, there is a large community around Battle Cats that has documented most levels on platforms like Fandom, specifically the Battle Cats Fandom Wiki. I developed a Python script to scrape this data, pulling descriptions, enemy types, recommended strategies, and other key details for each floor from 1 to 50.

Here's a breakdown of my code, which automates the process of retrieving and saving this information:

Explanation of Each Component

1. Modules and Libraries:
    - requests: This module lets me send HTTP requests to the Battle Cats Fandom Wiki and retrieve the HTML content of each page. Each floor in the Heavenly Tower has its own page, and the requests module lets me easily access this data.
    - BeautifulSoup (from bs4): This library is essential for parsing HTML. After receiving the raw HTML with requests, I use BeautifulSoup to extract readable text from the content.
    - os: The os module is used here to create directories and manage file paths. It ensures that all data is saved in a neatly organized way, with each floor's data stored as a separate text file.
2. Setting the Base URL:
    - The base URL (https://battle-cats.fandom.com/wiki/Heavenly_Tower/Floor_) provides the main structure for accessing each floor page. By appending the floor number to this base URL in a loop, I can access each individual floor's webpage.
    - os.makedirs(output_dir, exist_ok=True): This line checks if a folder called "heavenly_tower_floors" exists, and if not, it creates one. This is where all floor files will be stored, ensuring that my data stays organized.

3. Looping Through Each Floor:
    - for i in range(1, 51): This loop goes through each floor from 1 to 50, constructing the correct URL for each floor's page.
4. Fetching the Page Content:
    - response = requests.get(url): For each floor, this line sends a GET request to the Battle Cats Wiki and retrieves the HTML content.
    - if response.status_code != 200: This checks if the request was successful. If the request fails (status code other than 200), the script prints an error message and skips that floor.
5. Parsing the Content with BeautifulSoup:
    - soup = BeautifulSoup(response.content, "html.parser"): Here, BeautifulSoup processes the HTML content and allows me to extract only the readable text. This is important because raw HTML contains tags, links, and other data that isn't useful for a chatbot.
6. Extracting Relevant Text Only:
    - page_text = soup.get_text(separator="\n", strip=True): This line converts the parsed HTML into plain text.
    - if "Reference" in page_text: In some cases, there's additional data at the end of each page, starting with the word "Reference". By splitting the content at "Reference" and taking only the first part, I'm able to keep only the main content without any extra references or footnotes.
7. Saving the Data to Text Files:
    - Each floor's data is saved as a separate .txt file. This makes it easy to load each file into the chatbot later on, allowing it to answer questions floor by floor.

Why This Dataset Matters:

This dataset includes everything I need to train the chatbot on Heavenly Tower strategies. By parsing each floor's details into text files, I can now feed the data to the chatbot and have it provide strategic insights on-demand. This eliminates the need to visit multiple sources, and instead, I can simply ask the chatbot for tips and strategies specific to each floor. This is very helpful for me. Asking the chatbot information about upcoming stages makes it possible to beat the Heavenly Tower with much more efficiency.

This dataset also makes the chatbot practical and valuable to other Battle Cats players who might want a single resource for Heavenly Tower strategies.

3.1 Problem with the scraped data

Problem: Storing All Floors' Data in a Single File

Initially, I attempted to store all the floor data from the Heavenly Tower in a single file. This approach seemed logical because having everything in one place could simplify data loading. However, as I was testing the chatbot, it did not know specific information about a single floor. The neural network did not know where to access the necessary information in the large .txt formatted file. The large single file created inefficiencies in querying and indexing, as it contained all floors' data without any organization for individual access. This setup made it difficult to extract specific floor information accurately and efficiently.

Solution: Splitting Data by Floor

To resolve this, I decided to split the data into separate files for each floor. Organizing each floor's data in its own file allowed for more targeted access. Now, if a user queried for floor-specific information, your system could directly load that floor's file without scanning unrelated data. This change not only improved the speed and accuracy of the queries but also reduced memory consumption, as only relevant data was loaded for each query.

4.0 User Testing and Feedback

One of the key steps in evaluating the chatbot was to test it by asking a variety of queries. For instance, I asked the chatbot, *"Which enemies appear in Floor 30 of the Heavenly Tower?"* It responded with detailed information, including the specific enemy types. This response enabled me to plan an optimal strategy for tackling the level effectively.

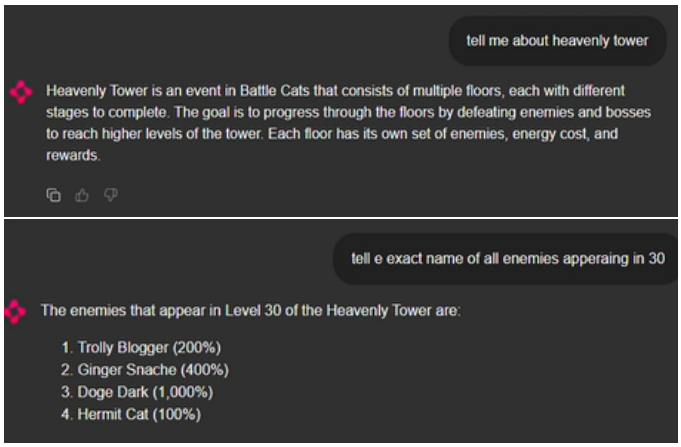
Even though the user input was no proper English, the query could still be understood. That is because the LLM knows how to handle with spelling errors like these.

The chatbot has demonstrated its versatility by providing accurate and helpful answers about various aspects of the game. Users can ask it about enemy appearances on specific levels, the buffs applied to enemies, the health of enemy bases, effective strategies for overcoming challenges, rewards earned by beating a level, and even details about the map size and layout. The chatbot's extensive knowledge base makes it a reliable companion for players seeking quick advice without having to search through multiple web pages.

However, user interactions also highlighted some areas for improvement. While the chatbot is already useful in its current form, there are limitations in its scope of knowledge and accessibility. Addressing these limitations would make the chatbot an even more valuable resource for players.

5.0 Future Improvements and Expansion Ideas

There are several ways to enhance the chatbot's functionality and usability to better serve players of *The Battle Cats*.



First, the chatbot's knowledge is currently based solely on data that was provided [2]. While this is a comprehensive and reliable resource, adding more data sources would make the chatbot even more versatile. For example, incorporating data from official game guides, community forums, or even experienced players' blogs and video tutorials could provide better of different responses to user queries.

Second, the chatbot is currently limited to localhost access, meaning only users on the same machine can use it. To broaden its reach, the chatbot could be made publicly accessible by deploying it to a cloud service like AWS, Azure, or Google Cloud. Alternatively, creating a web-based interface or a mobile app would allow players to access the chatbot from anywhere, making it a truly universal tool.

Third, the chatbot could be expanded to include information about levels beyond the Heavenly Tower. This would make it more useful to a broader audience of players. By covering additional content such as *Stories of Legend*, *Into the Future*, and *Cats of the Cosmos*, the chatbot could assist users in every aspect of the game.

Additionally, the chatbot could go beyond level-specific advice to include general game knowledge. It could guide players on which cats to prioritize upgrading based on utility, provide information on special events or offer tips for collecting treasures and maximizing bonuses. Expanding its scope in this way would help players improve their overall gameplay.

To make the chatbot even more user-friendly, context-aware suggestions for refining queries. These enhancements would make the chatbot easier to use for players of all experience levels.

Finally, integrating the chatbot with real-time game data would bring its functionality to the next level. For example, the chatbot could recommend strategies tailored to a user's current progress in the game, such as suggesting formations based on their available cats and upgrades. Additionally, it could automatically update its knowledge base to reflect new game content or patches, ensuring that

players always receive the most relevant advice.

By implementing these improvements, the chatbot could transform from a level-specific tool into a comprehensive and indispensable guide for all aspects of *The Battle Cats*. It would not only provide valuable advice for tackling the Heavenly Tower but also enhance the overall gaming experience for players around the world.

Conclusion

In conclusion, I chose *The Battle Cats* as an example for this chatbot project because it was simple yet effective for showing how a chatbot can work with specific data. I needed a dataset that wasn't already included in large AI models like ChatGPT. Details about levels, enemies, and strategies in *The Battle Cats* are not common knowledge for AI, which made it a great choice for this project.

However, this chatbot is not limited to just *The Battle Cats*. The same idea can be used for other games or even for company-specific data. For example, a company could create a chatbot to give helpful information about its products, services, or processes. This could save time, help customers get answers faster, and improve the way people work.

Having a chatbot like this has many advantages. It can give quick and accurate answers to users, so they don't have to search for information themselves. It is available at any time, making it very convenient. It can also handle repetitive questions, freeing up people to focus on more important tasks.

Another great thing about this kind of chatbot is that it can grow and improve. If you add more data or expand its knowledge, it can become even more useful. Whether it's for gaming, school, work, or other areas, custom chatbots can make life easier and help people in many ways.

This project shows how a chatbot can take specific data and make it easier to access and use. With more work and improvements, chatbots like this could be used to help people solve problems, save time, and be more productive in all kinds of areas.

6.0 References

[1] https://docs.llamaindex.ai/en/stable/understanding/putting_it_all_together/chatbots/building_a_chatbot/,

Provided code, accessed last: 12.11.2024

[2] https://battle-cats.fandom.com/wiki/Heavenly_Tower, Data which I scraped, accessed last: 12.11.2024

[3] https://classroom.google.com/u/1/c/NzE0MDU5NjM3MTQy, google classroom, accessed last: 12.11.2024

[4] https://docs.chainlit.io/backend/config/ui, dictionary on how to use chainlit, accessed last: 12.11.2024

[5] https://docs.literalai.com/get-started/overview, Literal AI website, accessed last: 12.11.2024

[6] https://platform.openai.com/docs/overview, Open AI API access, accessed last: 12.11.2024