Simple RAG Chatbot Project Report

Objective

The goal of this project is to create a specialized chatbot that uses a Large Language Model (LLM) for understanding and generating natural language, along with a vector database for storing and retrieving information. By using the Retrieval-Augmented Generation (RAG) method, the chatbot aims to give accurate and relevant responses. The front-end of this application will be built using Streamlit, a Python framework for interactive web apps.

Problems with Generic LLM

Generic LLMs have two main issues:

Lack of Knowledge: Sometimes, the model admits it doesn't know about a topic because it hasn't been trained on that specific data.

Hallucination: The model might give incorrect or misleading answers when it doesn't have enough information, especially about specialized topics like legal rules or medical data.

Introducing RAG

RAG (Retrieval-Augmented Generation) improves LLMs by adding more data to them. It has two main components:

Indexing: Organizing data from various sources so the system can easily use it.

Retrieval and Generation: The retrieval component acts like a search engine, finding relevant data related to the user's query. This data is then fed into the LLM, which uses this context to generate a more informed and accurate response.

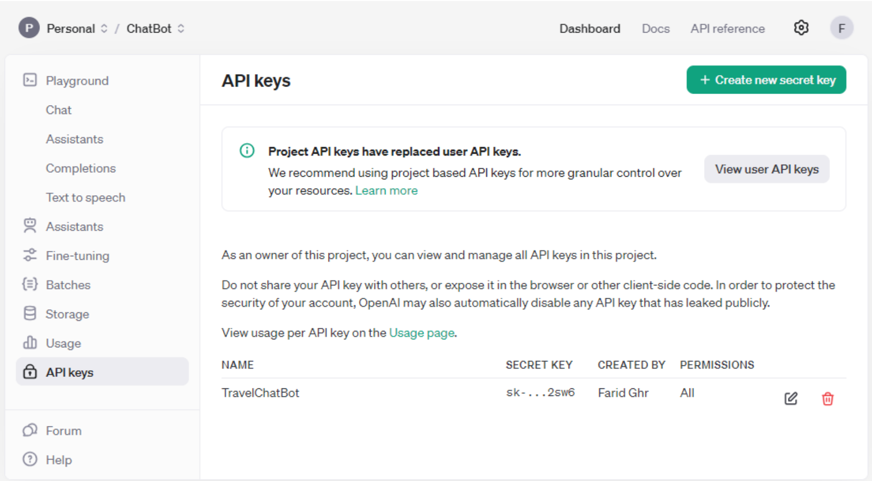
In simple terms, RAG helps LLMs give better answers by pulling in extra information when needed.

Technology Stack

- **Frontend:** Streamlit for building the user interface. Streamlit is a popular Python framework for creating interactive web applications.
- **Vector Database:** Pinecone for efficient data storage and retrieval. Pinecone is designed for managing and querying large amounts of vector data quickly.
- **LLM:** OpenAI's API for the language model. OpenAI provides state-of-the-art language models accessible via an API.
- **Backend:** LangChain framework utilizing the RAG method. LangChain helps in building applications that combine language models with other tools and databases.

Setting Up OpenAI API

1. Log into your OpenAI account and go to the OpenAI Platform.
2. Navigate to the API section.
3. Create a new API key by clicking '+ Create' new secret key.
4. Name your key and click the Create secret key button.
5. Copy the secret key and add it to a file named '.env'.



API keys

Project API keys have replaced user API keys.
We recommend using project based API keys for more granular control over your resources. Learn more

As an owner of this project, you can view and manage all API keys in this project.

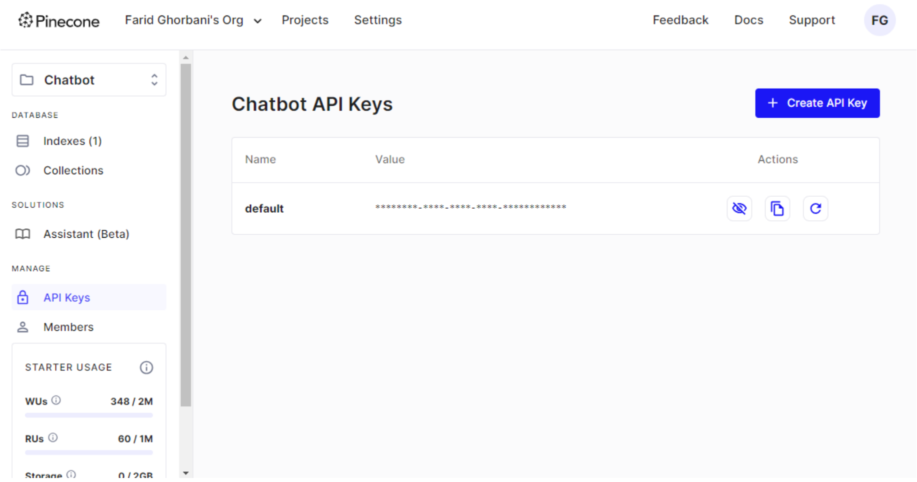
Do not share your API key with others, or expose it in the browser or other client-side code. In order to protect the security of your account, OpenAI may also automatically disable any API key that has leaked publicly.

View usage per API key on the Usage page.

NAME	SECRET KEY	CREATED BY	PERMISSIONS
TravelChatBot	sk-...2sw6	Farid Ghr	All

Setting Up Pinecone

1. Sign up for a Pinecone account here.
2. After registering, create a project by filling in the Project Name, Cloud Provider, and Environment.
3. Generate an API key in the API Keys section and save it securely.
4. Add your Pinecone API key to the .env file.

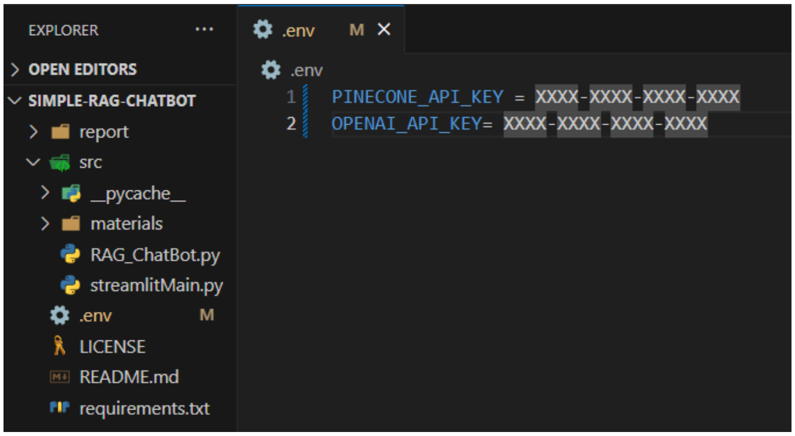


Chatbot API Keys

Name	Value	Actions
default	********-****-****-****-************	

Project Structure and Environment

After completing the account setup, you can create a directory called “Chatbot”. Inside the Chatbot directory, create a file called .env. The context inside .env should look like the image below (replace xxx with your OpenAI API Keys and Pinecone API Keys). We will be using this .env file in later steps.



```
PINECONE_API_KEY = XXXX-XXXX-XXXX-XXXX
OPENAI_API_KEY= XXXX-XXXX-XXXX-XXXX
```

In the one directory like 'src' create a file called RAG_ChatBot.py, and create an empty class called Chatbot inside it. This class is going to be called when we implement the frontend UI. For now, just create the class without adding any more code to it.

In the same directory create a file called streamlitMain.py, it used for frontend UI and we use our chatbot class in this file.

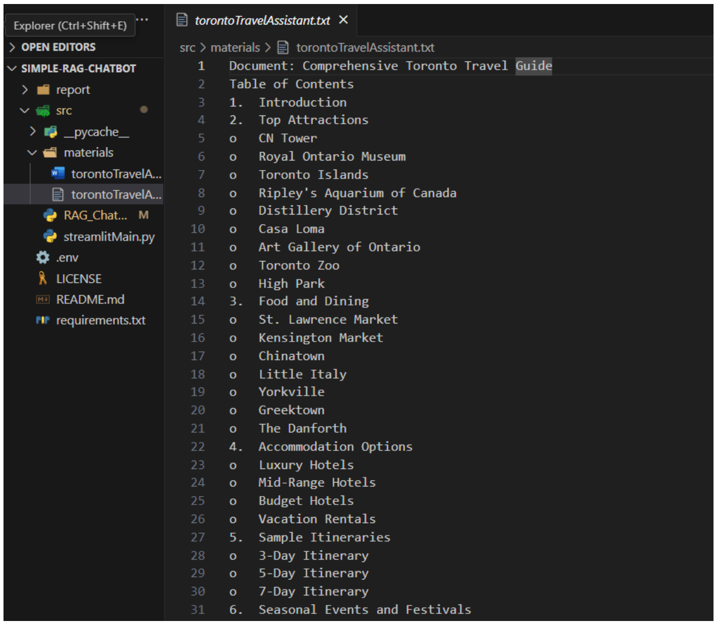
Importing Dependencies

Install the following dependencies:

- **langchain:** For using components and chains from the LangChain library.
- **pinecone-client:** To connect with Pinecone.
- **streamlit:** For creating UI pages with Python.
- **openai:** For interacting with OpenAI's API.
- **python-dotenv:** To use environment variables stored in .env.

Data indexing

Before we start indexing, we need to first have the data that our model will use to answer questions. In this case, we need a text file or a pdf about 'Toronto Travel Guide', as our chatbot will be answering questions related to this. in the 'src' directory, create a directory called 'materials' and put these files into it.



```
Document: Comprehensive Toronto Travel Guide
Table of Contents
1.  Introduction
2.  Top Attractions
o   CN Tower
o   Royal Ontario Museum
o   Toronto Islands
o   Ripley's Aquarium of Canada
o   Distillery District
o   Casa Loma
o   Art Gallery of Ontario
o   Toronto Zoo
o   High Park
3.  Food and Dining
o   St. Lawrence Market
o   Kensington Market
o   Chinatown
o   Little Italy
o   Yorkville
o   Greektown
o   The Danforth
4.  Accommodation Options
o   Luxury Hotels
o   Mid-Range Hotels
o   Budget Hotels
o   Vacation Rentals
5.  Sample Itineraries
o   3-Day Itinerary
o   5-Day Itinerary
o   7-Day Itinerary
6.  Seasonal Events and Festivals
```

Breaking Down Text Files

To make the text files manageable, we break them down into smaller segments using a text splitter. In this example, we set the chunk_size to 1000 and chunk_overlap to 4. This ensures that each segment is of a manageable size and overlaps slightly with the previous segment to maintain context.

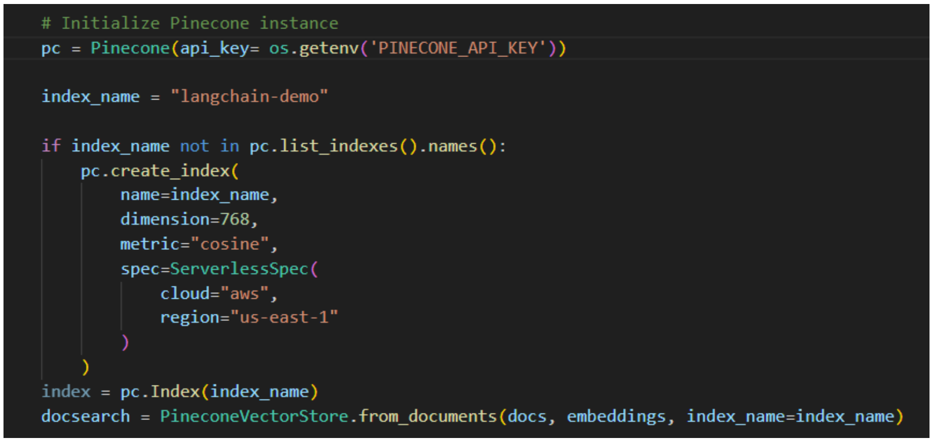
```
# Load and split documents
loader = TextLoader('./materials/torontoTravelAssistant.txt')
documents = loader.load()
text_splitter = CharacterTextSplitter(chunk_size=1000, chunk_overlap=4)
docs = text_splitter.split_documents(documents)

# Initialize embeddings
embeddings = HuggingFaceEmbeddings()
```

Embedding Text Segments

After splitting the text into segments, we use the HuggingFaceEmbedding tool to convert these segments into vector embeddings. These embeddings capture the semantic meaning of the text and are used for efficient retrieval from the vector database.

Storing Embeddings in Pinecone

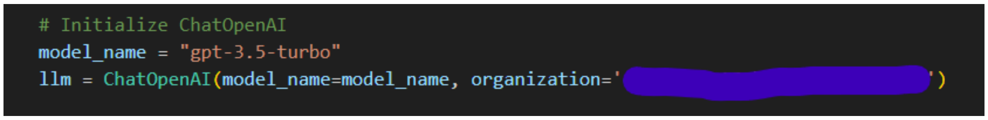
We initialize the Pinecone client in our application using the API key from the Pinecone dashboard. We then create an index in Pinecone to store the embedded text segments. If the index already exists, we update it with the new embeddings. This process ensures that our vector database is ready for efficient retrieval of relevant data during the chatbot's operation.



```
# Initialize Pinecone instance
pc = Pinecone(api_key= os.getenv('PINECONE_API_KEY'))

index_name = "langchain-demo"

if index_name not in pc.list_indexes().names():
    pc.create_index(
        name=index_name,
        dimension=768,
        metric="cosine",
        spec=ServerlessSpec(
            cloud="aws",
            region="us-east-1"
        )
    )
index = pc.Index(index_name)
docsearch = PineconeVectorStore.from_documents(docs, embeddings, index_name=index_name)
```

Model setup

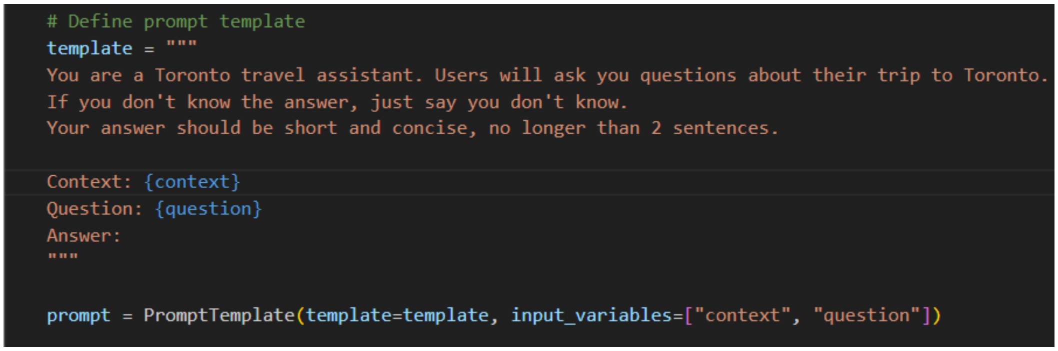
Now that we have our embedded texts on the vector database, let's move on to the model setup part. Of course, we don't want to create, train, and deploy the LLM from scratch locally. This is why we are using OpenAI.



```
# Initialize ChatOpenAI
model_name = "gpt-3.5-turbo"
llm = ChatOpenAI(model_name=model_name, organization='[redacted]')
```

Prompt engineering

For LLM to answer our question, we need to define a prompt that will contain all of the necessary information. This allows us to customise the model to fit our needs. In our case, we will tell the model to be a Toronto travel assistant and answer only relevant questions.Additionally, we need to pass {context} and {question} to the prompt. These values will be replaced with the data chunk we retrieve from our vector database for {context} and the question the user asked for the {question}.



```
# Define prompt template
template = """
You are a Toronto travel assistant. Users will ask you questions about their trip to Toronto.
If you don't know the answer, just say you don't know.
Your answer should be short and concise, no longer than 2 sentences.

Context: {context}
Question: {question}
Answer:
"""

prompt = PromptTemplate(template=template, input_variables=["context", "question"])
```

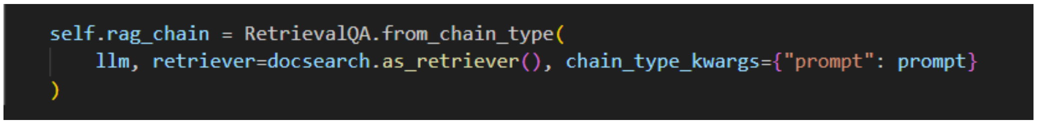
With this template created, we then define the PromptTemplate object taking our template and input variables (context and questions) as a parameter.

Chaining it all together

Now that we have:

1. Pinecone database index object(docsearch)
2. PromptTemplate (prompt)
3. Model (llm)

We are ready to chain them together. The process starts with docsearch pulling relevant documents to provide context. Next, a prompt step refines or modifies the query before it's processed by our model, llm.



```
self.rag_chain = RetrievalQA.from_chain_type(
    llm, retriever=docsearch.as_retriever(), chain_type_kwargs={"prompt": prompt}
)
```

Finalizing the Model

Implementing the Chatbot Class

In the RAG_ChatBot.py file, we define the Chatbot class and implement the RAG chain within this class. This class handles the entire process of retrieving relevant data, generating a prompt, and obtaining a response from the LLM. By encapsulating this functionality within a class, we make it easy to reuse and extend the chatbot's capabilities in the future.

Streamlit Frontend

Create the frontend in streamlitMain.py using Streamlit. This file will handle the UI and interact with the Chatbot class.

Creating the Streamlit Interface

In streamlitMain.py, we create a simple and intuitive interface using Streamlit. This interface allows users to interact with the chatbot by typing their queries and receiving responses. The frontend connects to the Chatbot class and displays the generated responses in real-time.

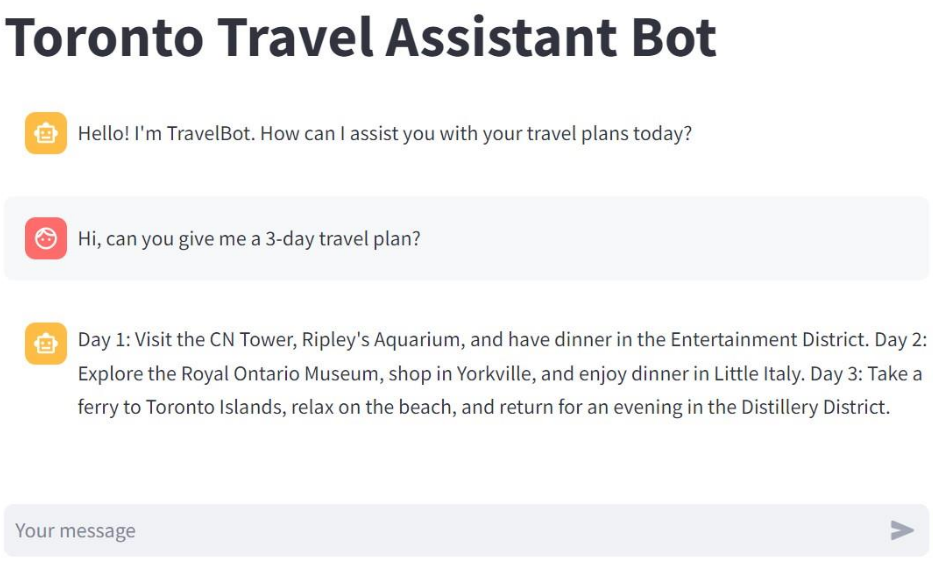
Running the Application

To run the application, use the following command: “streamlit run streamlitMain.py”

This command starts the Streamlit server and opens the chatbot interface in a web browser.

Result

As you can see, the bot can answer (or fortune tell) any question related to Toronto Travel Assistance. It also does not answer irrelevant questions, thanks to the prompt template we have used.

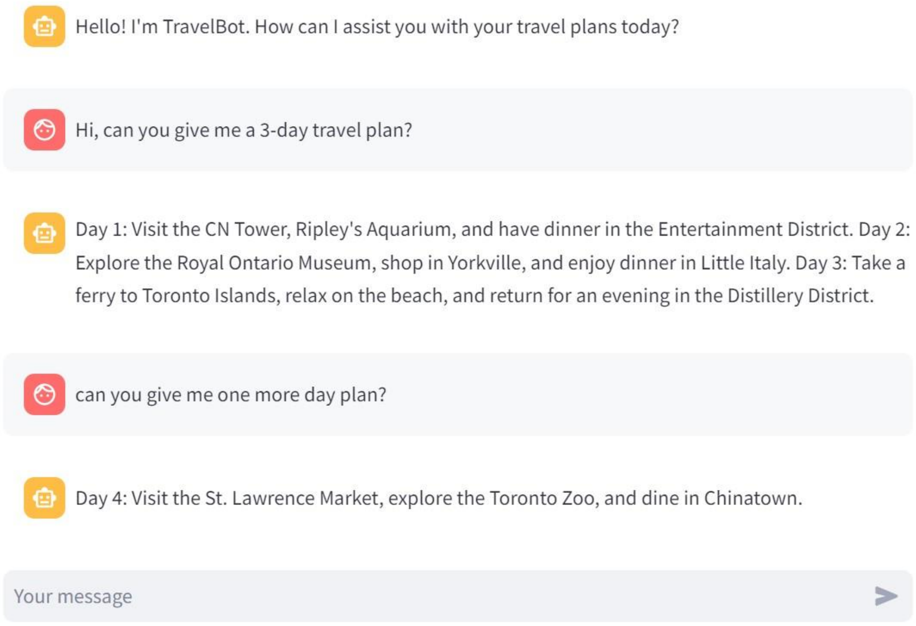


Toronto Travel Assistant Bot

Hello! I'm TravelBot. How can I assist you with your travel plans today?

Hi, can you give me a 3-day travel plan?

Day 1: Visit the CN Tower, Ripley's Aquarium, and have dinner in the Entertainment District. Day 2: Explore the Royal Ontario Museum, shop in Yorkville, and enjoy dinner in Little Italy. Day 3: Take a ferry to Toronto Islands, relax on the beach, and return for an evening in the Distillery District.



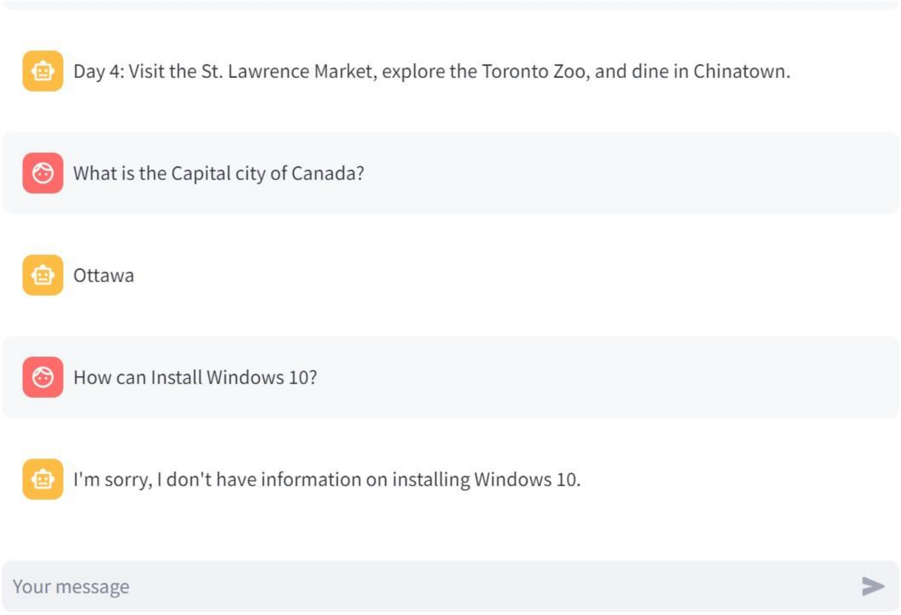
Hello! I'm TravelBot. How can I assist you with your travel plans today?

Hi, can you give me a 3-day travel plan?

Day 1: Visit the CN Tower, Ripley's Aquarium, and have dinner in the Entertainment District. Day 2: Explore the Royal Ontario Museum, shop in Yorkville, and enjoy dinner in Little Italy. Day 3: Take a ferry to Toronto Islands, relax on the beach, and return for an evening in the Distillery District.

can you give me one more day plan?

Day 4: Visit the St. Lawrence Market, explore the Toronto Zoo, and dine in Chinatown.



Day 4: Visit the St. Lawrence Market, explore the Toronto Zoo, and dine in Chinatown.

What is the Capital city of Canada?

Ottawa

How can Install Windows 10?

I'm sorry, I don't have information on installing Windows 10.

Conclusion

The model isn't perfect and there are areas for improvement. However, this project provides a basic understanding of creating a RAG chatbot and using vector databases.

Future Improvements

There are several ways to enhance the chatbot:

1. **Expand the Dataset:** Include more diverse and comprehensive data sources to improve the chatbot's knowledge base.
2. **Optimize the Prompt:** Refine the prompt template to handle more complex queries and

You can take a look at the code repository here in [my github repo](https://github.com/Faridghr/Simple-RAG-Chatbot).