AutoBuddy: A Car Shopping Search Chatbot

Farid Ghorbani
Ghorbani.f@northeastern.edu
College of Engineering
Northeastern University
Toronto, ON

Prompt Engineering & AI
Summer 2024

1. Introduction

1.1. Objective

AutoBuddy is a car shopping search chatbot designed to assist users in finding the perfect car based on their preferences, budget, and specific criteria. It leverages natural language processing and Retrieval-Augmented Generation (RAG) to provide a personalized and efficient car search experience.

In this project, we aim to develop a domain-specific chatbot application that utilizes a Large Language Model (LLM) for natural language understanding and processing, combined with the efficiency and scalability of a vector database for data storage and retrieval. The application will implement the Advanced Retrieval-Augmented Generation (RAG) method to enhance the chatbot's ability to provide accurate and relevant responses by integrating retrieved information with generative AI capabilities. We also fine-tune GPT-4o-mini in this project with related data to achieve optimal performance.

The front-end of the application will be developed using Streamlit, a popular Python framework for creating interactive web applications.

1.2. Detailed Description

AutoBuddy offers users a personalized car search experience by understanding their preferences and providing tailored recommendations. The chatbot handles user queries, searches through car listings, and provides detailed information on selected cars.

Key Features:

- **Personalized Car Recommendations** based on user preferences (e.g., budget, car type, brand).
- **Advanced Retrieval-Augmented Generation (RAG)** pipeline for contextually relevant responses.
- **Fine-Tuned LLM (GPT-4o-mini)** for precise, domain-specific car shopping insights.
- **Performance Evaluation** using context relevance and human-assessed answer relevance.
- **Interactive Interface** built with Streamlit for a user-friendly experience.
- **Scalable Vector Database** leveraging Pinecone for fast and efficient data retrieval.

1.3. Specific Problem

The project addresses the complexity and time-consuming nature of car shopping by offering a streamlined, AI-driven search tool that simplifies the decision-making process.

2. Project Architecture

2.1. Technologies Used

- **Programming Language**: Python
- **Natural Language Processing (NLP)**: OpenAI API
- **Query Processing and Data Retrieval**: LangChain and Retrieval-Augmented Generation (RAG)
- **Vector Database**: Pinecone
- **User Interface**: Streamlit

2.2. Data Collection and Preprocessing

Before indexing, we need to gather the data that our model will use to answer questions.

Source and Nature of Data: Data is collected from various online car company websites.
Steps for Data Collection: Web scraping from car listing websites, API integration.
Data Preprocessing Techniques: Data cleaning and feature extraction.

During the data collection phase, which was one of the most important phases of this project, I used web scraping methods to extract information from various online car company and dealer inventory websites.

First, I gathered a list of all available car models in Canada, totaling around 200 different models. For each model, I collected information through web scraping and API integration. After finishing data collection, I consolidated the information into a text file, preparing it for the next phase.

2.3. RAG Pipeline Implementation

2.3.1 Project Flow

Breaking Down Text Files

After collecting the information, we break it down into smaller segments using a text splitter to make the text files manageable. In this example, we set the chunk size to 1000 and chunk overlap to 4. This ensures that each segment is manageable and overlaps slightly with the previous segment to maintain context.

```
# Load and split documents
loader = TextLoader(material_file_path)
documents = loader.load()
text_splitter = CharacterTextSplitter(chunk_size=1500, chunk_overlap=300)
docs = text_splitter.split_documents(documents)
```

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 embeddings
embeddings = HuggingFaceEmbeddings()

# 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)
self.docsearch = PineconeVectorStore.from_documents(docs, embeddings, index_name=index_name)
```

2.3.2 Advanced RAG Technique

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

1. **Indexing**: Organizing data from various sources so the system can easily use it.
2. **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 this project, we focus on key index retrieval methods. The essence of the RAG system lies in its ability to store and retrieve vectorized content efficiently.

To improve my RAG pipeline and the performance of our chatbot, I employed advanced RAG techniques such as Auto-Merging Retrieval, which enhances the relevance of context retrieval. Auto-merging combines multiple retrieved contexts to generate a more comprehensive and relevant response. This approach is particularly useful when dealing with diverse or overlapping pieces of information.

Implementing Auto-Merging Retrieval

To incorporate auto-merging retrieval into your chatbot setup:

1. **Setup Auto-Merging Retrieval**: Modify the retrieval process to combine retrieved contexts before passing them to the language model. Adjust the handling of retrieved documents in the ConversationalRetrievalChain or RetrievalQA chain.
2. **Implement Auto-Merging Retrieval**: Here's an updated version of your chatbot code with auto-merging retrieval, including modifications to combine multiple retrieved contexts:

```
# Define a function for combining contexts
def merge_contexts(self, contexts):
    # Combine the contexts into a single string
    return "\n\n".join([ctx.page_content for ctx in contexts])

# Create a function to retrieve and merge contexts
def retrieve_and_merge_contexts(self, query, retriever):
    # Retrieve multiple contexts
    retrieved_docs = retriever.get_relevant_documents(query)
    # Merge contexts
    combined_context = self.merge_contexts(retrieved_docs)
    return combined_context
```

2.4. Prompt Engineering

To ensure that the LLM answers our questions correctly, we need to define a prompt that contains all the necessary information. This allows us to customize the model to fit our needs. In our case, we will include placeholders for {context} and {question} in the prompt. These placeholders will be replaced with the data chunk retrieved from our vector database for {context} and the user's query for {question}.

2.5. Fine-Tuning LLMs

2.5.1 Problems with Generic LLMs

Generic LLMs have two main issues:

- **Lack of Knowledge**: The model may admit 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 lacks sufficient information, especially about specialized topics like legal rules or medical data.

2.5.2 Fine-Tuning

Fine-tuning enhances the performance of ChatGPT models, offering:

- **Higher Quality Results**: Achieves better accuracy than just prompt adjustments.
- **Efficient Training**: Handles more extensive examples than prompt-based learning.
- **Cost and Latency Reduction**: Shorter prompts result in lower costs and faster responses.

Fine-Tuning Steps

1. **Prepare Training Data**: We create datasets representative of the desired tasks. Each record is structured to demonstrate different interactions for a chatbot that provides factual responses with a specific tone. This varied dataset helps train the model to recognize and adapt to various conversational cues and responses.
2. **Uploading a Training File for Fine-Tuning**: Uploading our training data is a crucial step. The Python script below can be used to upload your training file. Ensure your file is correctly named and located in the correct directory before running the script.

```
client = OpenAI(api_key = os.environ.get("OPENAI_API_KEY"))

response = client.files.create(
    file=open("./Data/Dataset.jsonl", "rb"),
    purpose="fine-tune"
)
print(response)
```

3. **Train a New Fine-Tuned Model**:
    - **Install the OpenAI SDK**: Ensure that the OpenAI Python library is installed to interact with the API.
    - **Create a Fine-Tuning Job**: Use the following Python code to start a fine-tuning job. You'll need the file ID of your uploaded training data.

```
client = OpenAI(api_key = os.environ.get("OPENAI_API_KEY"))

# Start a fine-tuning job
response = client.fine_tuning.jobs.create(
    training_file="file-JZPZQFX06F5wfkXJpmL29gXL",
    model="gpt-4o-mini-2024-07-18"  # Model type to fine-tune
)
print(response)
```

    - **Monitor Job Progress**: After initiating a fine-tuning job, it may take some time to complete. You can check the status of your job with the following code.

```
1  # Retrieve the state of a fine-tune job
2  job_status = client.fine_tuning.jobs.retrieve("ftjob-hZxbrQG31pxDDA16kruQV8F8")
3  print(job_status)
```

4. **Using a Fine-Tuned Model**: After completing the fine-tuning process, the newly fine-tuned model can be integrated into the AutoBuddy chatbot to enhance its performance. Here's how you can utilize the fine-tuned model:

   - **Installation of OpenAI SDK**: If not already installed, ensure the OpenAI Python library is set up to facilitate interaction with the API.
   - **Implementing the Fine-Tuned Model**: You can use the fine-tuned model by referencing its specific model ID within your application code. This allows the chatbot to generate more accurate and contextually relevant responses tailored to user queries.
   - **Example Usage**: Here is a basic example of how to use the fine-tuned model in a Python script:

```
client = OpenAI(api_key = os.environ.get("OPENAI_API_KEY"))

completion = client.chat.completions.create(
    model=fine_tuned_model_id,  # Replace with your specific model identifier
    messages=[
        {"role": "system", "content": "You are a helpful assistant."},
        {"role": "user", "content": "Hello!"}
    ]
)
print(completion.choices[0].message)
```

2.6. Chaining it All Together

Now that we have:

- **Document Retrieval**: The docsearch component pulls relevant documents from the Pinecone index to provide contextual information related to the user's query.
- **Prompt Structuring**: The retrieved context is structured with the PromptTemplate, which formats the query and context for processing by the LLM.
- **Response Generation**: The fine-tuned LLM processes the structured prompt and generates a relevant response.

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.

```
# Create a RetrievalQA chain with the refined prompt and custom retrieval
def generate_response(self, question):
    # Retrieve and merge context
    combined_context = self.retrieve_and_merge_contexts(question, self.docsearch.as_retriever())
    # Format the prompt
    formatted_prompt = self.prompt.format(context=combined_context, question=question)

    # Generate response
    response = self.llm.invoke(formatted_prompt)
    return response.content
```

2.7. User Interface with Streamlit

The front-end of AutoBuddy is built using Streamlit, which provides an interactive interface for users to interact with the chatbot. Streamlit is chosen for its simplicity and the ability to quickly prototype and deploy web applications.

Key Features:

- **Interactive Query Input**: Users can input their car search queries directly into the chatbot interface.
- **Display of Search Results**: The results retrieved and generated by the LLM are displayed in a user-friendly format, including details such as car models, prices, features, and dealer information.
- **User Feedback Mechanism**: Users can provide feedback on the responses, which can be used to further fine-tune the model or improve the retrieval process.

3. Evaluate Performance

The performance of a Retrieval-Augmented Generation (RAG) pipeline can be evaluated by focusing on its three main components: query, context, and response. The query represents the user's input, the context refers to the retrieved information that aligns with the query, and the response is the final output generated by the Large Language Model (LLM) based on both the query and context. We assess the RAG pipeline using two key metrics: Context Relevance and Answer Relevance.

To conduct the evaluation, we utilize a Python notebook for human assessment of Answer Relevance and cosine similarity for Context Relevance.

3.1. Evaluating Context Relevance

Context Relevance is critical for ensuring that the information retrieved by the system is pertinent to the user's query. To evaluate Context Relevance, we follow these steps:

1. **Query Generation**: We generate a broad set of prompts to cover a wide range of queries, ensuring that the evaluation process is comprehensive.

```
queries = [
    "What's the best electric car for long-distance travel?",
    "Can you suggest a reliable SUV for a family of four?",
    "What's a good car with a spacious trunk?",
    "Which hybrid car offers the best fuel efficiency?",
    "Can you recommend a compact car that's easy to park in the city?"
]
```

2. **Context Retrieval**: For each query, we retrieve the context using the RAG pipeline. This context is then assessed for relevance using cosine similarity.

3. **Text Vectorization**: We use Term Frequency-Inverse Document Frequency (TF-IDF) to convert the texts (both questions and retrieved contexts) into numerical vectors, which allows for easier comparison of the importance of words in each document.

```
def evaluate_context_relevance(questions, retrieved_contexts):
    vectorizer = TfidfVectorizer().fit_transform(retrieved_contexts + questions)
    vectors = vectorizer.toarray()
    relevance_scores = []

    for i, question in enumerate(questions):
        context_vector = vectors[i]
        question_vector = vectors[len(questions) + i]
        cosine_sim = cosine_similarity([context_vector], [question_vector])[0][0]
        relevance_scores.append(cosine_sim)

    return relevance_scores
```

4. **Cosine Similarity Calculation**: Cosine similarity is employed to measure the cosine of the angle between two vectors, representing the degree of similarity between them. For each question, we calculate the cosine similarity between the TF-IDF vector of the question and the corresponding retrieved context. Higher cosine similarity scores indicate a higher relevance of the retrieved context to the query.

```
[0.10596395744140033,
 0.020952840189126704,
 0.0,
 0.1951946798984531,
 0.03682295724389945]
```

3.2. Evaluating Answer Relevance

Answer Relevance assesses the accuracy and usefulness of the responses generated by the chatbot. The evaluation process involves the following steps:

1. **Context Retrieval and Answer Generation**: For each query, the RAG pipeline generates a context retrieval and a chatbot answer. These are stored together in a JSON file to facilitate analysis.
2. **Human Assessment**: The answers are evaluated through human assessment to determine how well they address the user's queries.
3. **User Feedback**: User feedback is also collected and analyzed as part of the Answer Relevance evaluation. This feedback helps identify areas where the chatbot's responses may need improvement.

4. Future Work

AutoBuddy is an evolving project with several areas identified for future enhancement:

- **Integration with More Car Listing Platforms**: Expanding the data sources to include more car listing platforms will provide users with a broader range of options.
- **Advanced Search Filters**: Implementing more refined search filters, such as by specific car features or dealer locations, will enhance the user experience.
- **Voice Interaction Capabilities**: Introducing voice interaction will make the chatbot more accessible and user-friendly, especially for mobile users.
- **Expansion to Other Markets**: Expanding the chatbot's capabilities to cover additional markets, such as used cars and electric vehicles, will attract a wider audience.

5. Conclusion

In this project, we developed AutoBuddy, a car shopping assistant chatbot that leverages advanced Retrieval-Augmented Generation (RAG) techniques and a fine-tuned Large Language Model (LLM) to deliver personalized and efficient car search experiences. Through rigorous performance evaluation in both Context Relevance and Answer Relevance, we demonstrated the chatbot's ability to provide accurate, relevant, and user-friendly responses.

AutoBuddy represents a significant step forward in the use of AI for simplifying complex decision-making processes in the car shopping industry. As we continue to refine the chatbot and expand its capabilities, we anticipate that AutoBuddy will become an indispensable tool for car shoppers seeking tailored recommendations and a streamlined purchasing experience.