Bookino

Book Recommendation Chatbot Project

Shima Bolboli

Summer 2024

Introduction

The Book Recommendation Chatbot is designed to assist users in finding book recommendations based on their queries. Leveraging advanced natural language processing (NLP) and machine learning techniques, this project utilizes LangChain, Pinecone,RAG, and OpenAI's fine-tuned models to deliver personalized and relevant book suggestions.

Objectives

- **Provide Accurate Book Recommendations:** To recommend books based on user queries by processing and understanding the context of the request.
- **Leverage Generative AI:** Use a fine-tuned model to generate coherent and contextually appropriate responses.
- **Utilize Retrieval-Augmented Generation (RAG):** Incorporate relevant context from a database to enhance the quality of recommendations.

Detailed Explanation of the Use Case

How It Works

1. **Document Preparation and Loading:**

   - **Data Set**: Initially, the books dataset from Kaggle.com was in CSV format. Due to long response delays when uploading this file directly to Pinecone, it was converted to a text file using the Pandas library. The converted text file was then indexed in Pinecone.

   - **TextLoader:** Loads the book-related text documents from a source file.

     ```
     loader = TextLoader('../book.txt')
     ```

   - **CharacterTextSplitter:** Splits the text into manageable chunks to facilitate efficient processing. If I did not chunk the data all the file was indexed in to one record in database and because the size of my dataset file was bigger than the maximum size of each record in Pinecone I got error. So I chunk the data and I try to find a balance between the chunk_size and quality.

     ```
     text_splitter = CharacterTextSplitter(chunk_size=1000, chunk_overlap=100)
     ```

2. **Embedding and Indexing:**

- **HuggingFaceEmbeddings:** Converts text chunks into vector embeddings using a HuggingFace pre-trained embedding model (`sentence-transformers/all-MiniLM-L6-v2`).

```
embeddings = HuggingFaceEmbeddings(model_name="sentence-transformers/all-MiniLM-L6-v2")
```

- **Pinecone:** Utilizes Pinecone for indexing and vector search. An index is created or checked for existing entries to store and retrieve document vectors.

```
# Initialize Pinecone instance
pc = Pinecone(api_key=pinecone_api_key, environment='us-east-1')
index_name = "langchain-demo"

# Check if index exists
if index_name not in pc.list_indexes().names():
    pc.create_index(
        name=index_name,
        dimension=384,
        metric="cosine",
        spec=ServerlessSpec(
            cloud="aws",
            region="us-east-1"
        )
    )
index = pc.Index(index_name)
```

3. Contextual Retrieval:

- **PineconeVectorStore:** Retrieves relevant documents based on user queries using the vector search capability.

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

4. Prompt Engineering and Response Generation:

- **PromptTemplate:** Defines a structured prompt to provide context for generating responses. In this step I tried different template and the content of this directly related to the quality of generated response.

```
template = """
    You are a helpful assistant who provides book recommendations based on user queries.
    Answer the question in your own words only from the context given to you.
    If questions are asked where there is no relevant context available, please ask the user to ask relevant questions.

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

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

5. Fine_tunning Process

Fine-tuning improves on few-shot learning by training on many more examples than can fit in the prompt, letting you achieve better results on a wide number of tasks. Once a model has been fine-tuned, you won't need to provide as many examples in the prompt. This saves costs and enables lower-latency requests.

1. **Jsonl file:** Prepares data for training the model by creating a diverse set of demonstration conversations. Each JSONL entry consists of messages exchanged between the user and the assistant, including system, user, and assistant messages. For example:

   System Message: Establishes the chatbot's role as a book recommendation assistant.

```
{"messages": [{"role": "system", "content": "You are a helpful assistant who
provides book recommendations."},
```

   User Query: The user seeks a recommendation for a mystery book

```
{"role": "user", "content": "What is 'The Catcher in the Rye' about?"},
```

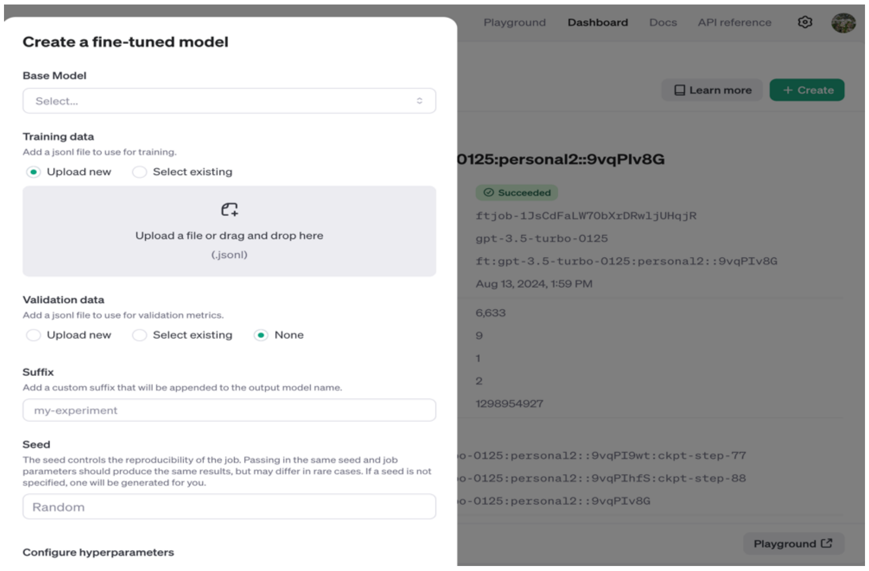
   Assistant Response: Provides a relevant book recommendation based on the user's query.

```
{"role": "assistant", "content": "'The Catcher in the Rye' by J.D. Salinger is
a novel that explores themes of teenage angst, alienation, and the challenges
of growing up through the eyes of its protagonist, Holden Caulfield."}]}
```

   The JSONL entries illustrate the chatbot's ability to handle a variety of book-related queries by generating contextually appropriate responses. Each entry is crafted to guide the fine-tuned model in providing accurate and useful recommendations or information about books. These interactions are crucial for training the model to understand and respond effectively to user queries, ensuring a high-quality user experience. I need at least 10 examples.

2. **Upload training data:** we can do this step in 2 ways:

   2.1. Upload the jsonl file in https://platform.openai.com/finetune



Create a fine-tuned model

Base Model
Select...

Training data
Add a jsonl file to use for training.
[✓ Upload new] Select existing

Upload a file or drag and drop here
(.jsonl)

Validation data
Add a jsonl file to use for validation metrics.
Upload new Select existing [✓ None]

Suffix
Add a custom suffix that will be appended to the output model name.
my-experiment

Seed
The seed controls the reproducibility of the job. Passing in the same seed and job parameters should produce the same results, but may differ in rare cases. If a seed is not specified, one will be generated for you.
Random

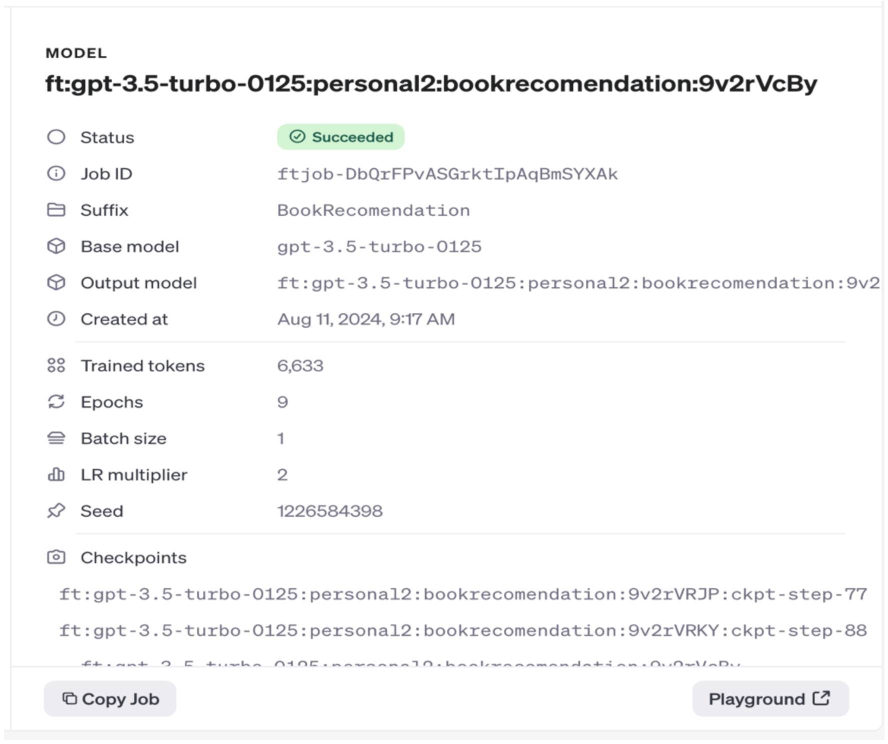
Configure hyperparameters

0125:personal2::9vqPIv8G

Succeeded
ftjob-1JsCdFaLW7ObXrDRwljUHqjR
gpt-3.5-turbo-0125
ft:gpt-3.5-turbo-0125:personal2::9vqPIv8G
Aug 13, 2024, 1:59 PM

6,633
9
1
2
1298954927

o-0125:personal2::9vqPI9wt:ckpt-step-77
o-0125:personal2::9vqPIhfS:ckpt-step-88
o-0125:personal2::9vqPIv8G



MODEL
ft:gpt-3.5-turbo-0125:personal2:bookrecomendation:9v2rVcBy

Status	Succeeded
Job ID	ftjob-DbQrFPvASGrktIpAqBmSYXAk
Suffix	BookRecomendation
Base model	gpt-3.5-turbo-0125
Output model	ft:gpt-3.5-turbo-0125:personal2:bookrecomendation:9v2
Created at	Aug 11, 2024, 9:17 AM
Trained tokens	6,633
Epochs	9
Batch size	1
LR multiplier	2
Seed	1226584398

Checkpoints
ft:gpt-3.5-turbo-0125:personal2:bookrecomendation:9v2rVRJP:ckpt-step-77
ft:gpt-3.5-turbo-0125:personal2:bookrecomendation:9v2rVRKY:ckpt-step-88

2.2. Use the following code to upload fine-tune data:

Check for Existing Fine-Tuned Models:

- Retrieves a list of fine-tuning jobs using `openai.FineTuningJob.list()`.
- Iterates through the jobs to find one with a status of `'succeeded'`.
- If a successful job is found, it uses the fine-tuned model ID from that job.

```
# Check if a fine-tuned model already exists
fine_tuned_model_id = None
fine_tuning_jobs = openai.FineTuningJob.list()

for job in fine_tuning_jobs['data']:
    if job['status'] == 'succeeded' and job.get('fine_tuned_model'):
        fine_tuned_model_id = job['fine_tuned_model']
        print(f"Using existing fine-tuned model: {fine_tuned_model_id}")
        break
```

Create a New Fine-Tuned Model:

- If no existing fine-tuned model is found, the script uploads a new dataset (`Dataset.jsonl`) using `openai.File.create()`.
- The response includes a file ID, which is printed for reference.

```
response = openai.File.create(file=open("Dataset.jsonl", "rb"), purpose="fine-
tune")
```

Start Fine-Tuning:

- Initiates a fine-tuning job using the uploaded file by calling `openai.FineTuningJob.create()`.
- Specifies the base model (`"gpt-3.5-turbo"`) for fine-tuning and prints the job ID.

```
 fine_tune_response = openai.FineTuningJob.create(training_file=file_id,
model="gpt-3.5-turbo")
  fine_tune_job_id = fine_tune_response["id"]
```

Monitor Fine-Tuning Job:

- Continuously checks the status of the fine-tuning job using `openai.FineTuningJob.retrieve()`.
- Prints the current job status and checks if the job is completed (`'succeeded'`).
- If the job fails or is canceled, an error is raised.
- Pauses for 30 seconds between status checks to avoid excessive API calls.

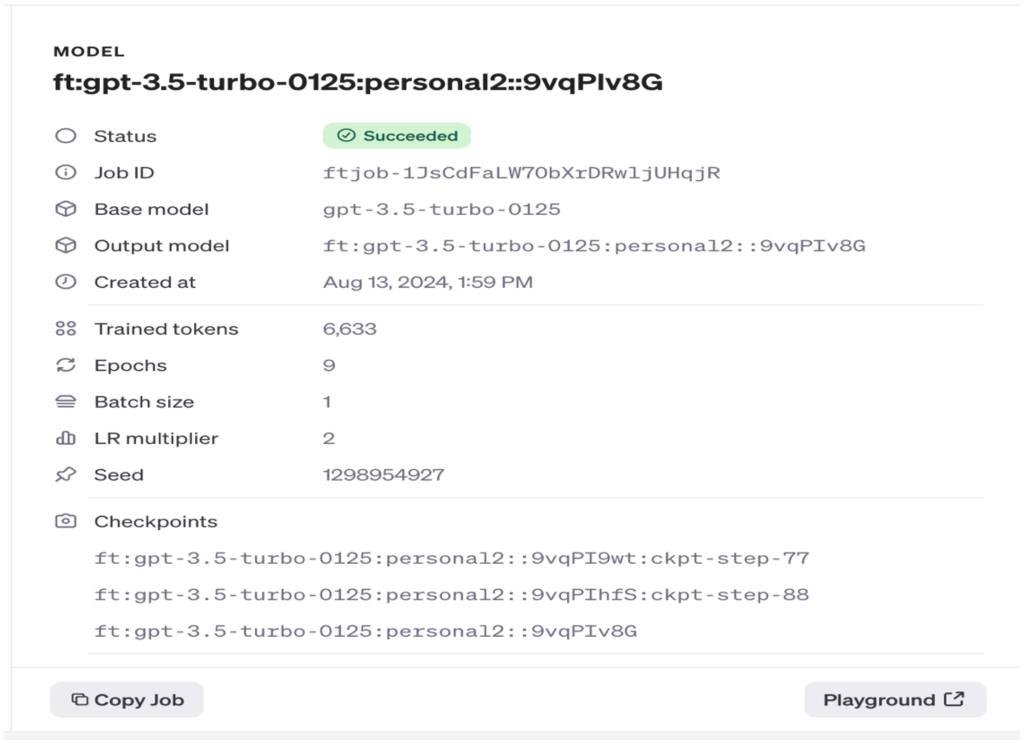
```
# Wait for fine-tuning to complete
    while True:
        job_status = openai.FineTuningJob.retrieve(fine_tune_job_id)
        print("Job Status:", job_status)
        if job_status['status'] == 'succeeded':
            fine_tuned_model_id = job_status['fine_tuned_model']
            print(f"Fine-tuned model created: {fine_tuned_model_id}")
            break
        elif job_status['status'] in ['failed', 'cancelled']:
            raise ValueError(f"Fine-tuning job failed or was cancelled:
{job_status}")
      time.sleep(30)  # Wait before checking again
```

Save Model ID:

- After successful fine-tuning, saves the fine-tuned model ID to a file (`fine_tuned_model_id.txt`).
- Prints a confirmation message.

```
# Save the fine-tuned model ID to a file
with open("fine_tuned_model_id.txt", "w") as f:
    f.write(fine_tuned_model_id)
    print(f"Fine-tuned model ID saved to fine_tuned_model_id.txt")
```

The result is like this:



MODEL
ft:gpt-3.5-turbo-0125:personal2::9vqPIv8G

Status	Succeeded
Job ID	ftjob-1JsCdFaLW70bXrDRwljUHqjR
Base model	gpt-3.5-turbo-0125
Output model	ft:gpt-3.5-turbo-0125:personal2::9vqPIv8G
Created at	Aug 13, 2024, 1:59 PM
Trained tokens	6,633
Epochs	9
Batch size	1
LR multiplier	2
Seed	1298954927

Checkpoints

```
ft:gpt-3.5-turbo-0125:personal2::9vqPI9wt:ckpt-step-77
ft:gpt-3.5-turbo-0125:personal2::9vqPIhfS:ckpt-step-88
ft:gpt-3.5-turbo-0125:personal2::9vqPIv8G
```

6. OpenAI API: Utilizes OpenAI's fine-tuned model to generate responses based on the retrieved context. If no relevant context is found, the model prompts the user to refine their query

```
def generate_response(question):
    # Retrieve relevant context from Pinecone
    search_results = docsearch.similarity_search(question, k=3)
    context = "\n".join([result.page_content for result in search_results])
    # Use the context in the prompt
    formatted_prompt = prompt.format(context=context, question=question)

    response = openai.ChatCompletion.create(
        model=fine_tuned_model_id,
        messages=[
     {"role": "system", "content": "You are a helpful assistant who provides
                  book recommendations based on user queries."},
            {"role": "user", "content": formatted_prompt.strip()}
        ],
        max_tokens=150,
        temperature=0.4
    )

    response_text = response['choices'][0]['message']['content'].strip()
```

7. Streamlit Interface:

- **Streamlit:** Provides a user-friendly interface for interaction. Users can input queries and receive recommendations through a web-based chat interface.

Key Features and Functionalities

- **Personalized Recommendations:** The chatbot provides book recommendations tailored to user queries.
- **Contextual Understanding:** The system uses context from a database of book-related documents to generate accurate responses.
- **Interactive User Interface:** Streamlit allows users to interact with the chatbot in a conversational manner.
- **Fine-Tuned Model:** Utilizes a fine-tuned OpenAI model to generate high-quality responses based on user input.

Challenges and Solutions

1. **Handling Large Datasets:**

- **Challenge:** Efficiently processing and indexing a vast amount of book data can be complex and resource-intensive.

- **Solution:** To address this, the dataset was converted from CSV format to a text file, which simplified the data handling process. Text splitting was employed to break down large documents into manageable chunks, and embeddings were used to encode and index the text efficiently. This approach allowed for scalable data management and retrieval.

2. **Ensuring Accurate Recommendations:**

- **Challenge:** Providing precise and relevant book recommendations based on user queries required contextual understanding.
- **Solution:** Implemented Retrieval-Augmented Generation (RAG) to enhance response accuracy. By integrating RAG, the system could retrieve and utilize relevant context from the document database, leading to more accurate and contextually appropriate recommendations.

3. **Integration and Deployment:**

- **Challenge:** Integrating various components, including Pinecone, LangChain, and OpenAI, to work seamlessly together.
- **Solution:** Utilized the PineconeVectorStore to bridge LangChain and Pinecone for efficient document retrieval. Proper configuration of API keys and environment settings ensured smooth operation and reliable interaction between the components.

Conclusion and Future Scope

The Book Recommendation Chatbot demonstrates how generative AI, RAG, and advanced NLP techniques can be employed to enhance user experiences in obtaining book recommendations. Future improvements could include:

- **Enhanced Model Fine-Tuning:** Continuously updating the fine-tuned model with new data to improve response quality.
- **Expanded Dataset:** Incorporating additional book data and genres to broaden the scope of recommendations.
- **User Feedback Integration:** Implementing a feedback mechanism to refine and adapt recommendations based on user interactions.

This project lays a solid foundation for building intelligent recommendation systems and can be extended to various domains beyond book recommendations.