2024

Chatbot for Financial and Operational Risk Guidelines

Capstone Project – Generative AI

DIBOSH BARUAH

Problem Statement:

The objective of this project is to build a chatbot using Langchain and Retrieval-Augmented Generation (RAG) techniques. The chatbot will be designed to answer questions related to financial and operational risk guidelines provided by the Reserve Bank of India (RBI). These documents contain important information about financial and operational risks that banks and financial institutions must adhere to. The chatbot will assist users in navigating and understanding the complex regulations and guidelines set forth by the RBI, facilitating improved comprehension and decision-making.

Solution Approach:

The solution is built around several key steps:

1. Document Loading and Preprocessing:

- **Input Data:** The input for this chatbot is comprised of two documents: "financial risk.pdf" and "operations risk.pdf", both containing RBI's guidelines.
- **Document Loading:** The documents are loaded using Langchain's PyPDFLoader. Each document is parsed and converted into structured text.
- **Text Splitting:** Once loaded, the document content is split into smaller chunks using Langchain's RecursiveCharacterTextSplitter to ensure efficient processing. Each chunk is no longer than 1000 characters with an overlap of 100 characters to retain context across chunks.

2. Embeddings Generation:

- **Text Embeddings:** The documents need to be converted into numerical representations (embeddings) for efficient retrieval during question answering. Initially, embeddings are generated using the DistilBERT model for each document chunk.

- **Model Used:** distilbert-base-uncased is used for generating embeddings from the documents, which are then stored as NumPy arrays.
- **Alternative Model:** The embeddings are further enhanced using the Hugging Face all-MiniLM-L6-v2 model, which is more suited for sentence-level embeddings and is integrated with Langchain's vector store (FAISS).

3. Storing Embeddings in a Vector Store:

- **Vector Store Creation:** The generated embeddings are stored in a FAISS (Facebook AI Similarity Search) vector store, which is used for fast similarity searches. This allows the chatbot to find the most relevant document chunks in response to a user query.
- **Saving the FAISS Index:** The FAISS index is saved locally for later retrieval during inference.

4. Summarization:

- **Context Summarization:** In cases where the document chunks exceed the maximum length for effective processing, the context is summarized using Hugging Face's facebook/bart-large-cnn model. The summarizer reduces the text to a concise form while retaining key information.
- **Contextual Representation:** This summarization ensures that even large documents can be represented in a way that is suitable for efficient querying and retrieval.

5. Question-Answering Pipeline:

- **Question-Answering (QA):** A pre-trained question-answering pipeline is used for processing user queries. The distilbert-base-cased-distilled-squad model is used for extracting answers from the summarized context.
- **Pipeline Flow:** The chatbot uses the vector store to search for the most relevant chunks and then feeds these chunks to the QA model to generate an answer based on the user's question.

6. FastAPI Integration for Serving the Model:

- **API Creation:** The FastAPI framework is used to create an API endpoint that allows users to interact with the chatbot. A Pydantic model is used to handle the incoming questions as part of the API request.
- **Running FastAPI Server:** The FastAPI app is run in a separate thread using uvicorn, which serves the chatbot model for real-time inference.
- **Similarity Search:** Each incoming query is used to perform a similarity search on the FAISS vector store to retrieve relevant document chunks that can be used as context for generating an answer.
- **Response Generation:** The chatbot then generates and returns an answer based on the context retrieved from the vector store.

7. Testing the API:

- **Client for API Testing:** A simple Python script is used to test the FastAPI server by sending HTTP POST requests with a user query. The response from the chatbot is then printed to verify the model's performance in answering real-world queries.

Key Insights:

- **Document Splitting:** By splitting documents into manageable chunks, the system ensures that it can handle large documents without overwhelming the model's input limits. This step is critical for ensuring scalability when working with lengthy financial regulations.
- **Embedding Models:** The choice of using a combination of DistilBERT and all-MiniLM-L6-v2 ensures a balance between computational efficiency and the quality of generated embeddings, providing a high level of accuracy in similarity searches.

- **Summarization:** Summarizing documents is essential to handle long texts efficiently. By leveraging the facebook/bart-large-cnn model, the system can ensure that users can receive relevant answers even from complex documents with large content.
- **FastAPI:** Integrating FastAPI enables the chatbot to be served as a scalable API, allowing users to interact with the model through a simple web interface. This makes the chatbot easily accessible and usable for a wide range of queries.

Future Applications:

- **Dynamic Update of Knowledge Base:** As new RBI guidelines are released, they can be easily incorporated into the system by reloading the documents, re-generating embeddings, and updating the FAISS vector store. This makes the system highly adaptable to future changes in regulations.
- **Scalability for Other Regulatory Documents:** The same architecture can be extended to other regulatory frameworks (e.g., SEBI, RBI, IRDAI) for creating similar chatbots for different sectors, enhancing the versatility of the solution.
- **Automated Compliance Checking:** The chatbot can be enhanced to assist financial institutions in ensuring compliance with RBI regulations by checking user-submitted queries against the relevant sections of the guidelines.

Conclusion:

This project successfully demonstrates the use of a chatbot to assist users in navigating financial and operational risk guidelines from the RBI. By leveraging Langchain’s document loaders, embeddings generation, and FastAPI integration, the chatbot can efficiently respond to complex queries, providing users with an intelligent, automated assistant to understand and interpret RBI regulations. The solution is scalable and adaptable, making it suitable for future updates and the addition of other regulatory frameworks.

8.Original Code:

```
# Original coding process
from langchain.document_loaders import PyPDFLoader

# File path for the RBI documents
pdf_paths = [
    "/Users/diboshbaruah/Desktop/Capstone_Project - Generative AI/financial
risk.pdf",
    "/Users/diboshbaruah/Desktop/Capstone_Project - Generative AI/operations
risk.pdf"
]

documents = []
for path in pdf_paths:
    loader = PyPDFLoader(path)
    documents.extend(loader.load())

# Initializing the RecursiveCharacterTextSplitter to break down text into
smaller chunks
from langchain.text_splitter import RecursiveCharacterTextSplitter

# Splitting the documents into smaller chunks
text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000,
chunk_overlap=100)
docs = text_splitter.split_documents(documents)

# Generating embeddings for each document chunk using DistilBERT
from transformers import DistilBertTokenizer, DistilBertModel
import torch
import numpy as np

# Loading the tokenizer and model
tokenizer = DistilBertTokenizer.from_pretrained("distilbert-base-uncased")
model = DistilBertModel.from_pretrained("distilbert-base-uncased")

# Converting the documents to embeddings directly
doc_embeddings = []

for doc in docs:
    # Tokenizing the document text
    inputs = tokenizer(doc.page_content, return_tensors="pt",
truncation=True, padding=True, max_length=512)

    # Getting the model outputs without calculating gradients
    with torch.no_grad():
        outputs = model(**inputs)

    # Getting the mean of the last hidden state (this gives us the embedding)

doc_embeddings.append(outputs.last_hidden_state.mean(dim=1).squeeze().numpy()
```

```
)

# Converting the list of embeddings to a NumPy array
doc_embeddings_np = np.array(doc_embeddings)

# Using Hugging Face's all-MiniLM-L6-v2 to create embeddings for document
content
from langchain_huggingface import HuggingFaceEmbeddings
from langchain.vectorstores import FAISS
from langchain.schema import Document

# Initializing the HuggingFaceEmbeddings model with a sentence-transformers
model
embedding_model = HuggingFaceEmbeddings(model_name="all-MiniLM-L6-v2")

# Sample documents
docs = [
    Document(page_content="This is a financial risk management document."),
    Document(page_content="This is an operational risk management
document."),
]

# Creating a FAISS vector store with the documents and embeddings model
vector_store = FAISS.from_documents(docs, embedding_model)

# Saving the FAISS index
vector_store.save_local("faiss_index")

from transformers import pipeline

summarizer = pipeline("summarization", model="facebook/bart-large-cnn")

# Extract context from document chunks
context = " ".join([doc.page_content for doc in docs])  # Combine all chunks
into a single context

max_context_length = 1000

if len(context.split()) > max_context_length:
    summarized_context = summarizer(context, max_length=300, min_length=100,
do_sample=False)[0]['summary_text']
else:
    summarized_context = context

# 6. Question-Answering Pipeline
qa_pipeline = pipeline("question-answering", model="distilbert-base-cased-
distilled-squad")

query = "What are the main types of risks discussed in the documents?"

answer = qa_pipeline(question=query, context=summarized_context)
```

```
# Print the Answer
print(answer)
```

```
Device set to use mps:0
Device set to use mps:0
```

```
{'score': 0.2168053239583969, 'start': 57, 'end': 84, 'answer': 'operational
risk management'}
```

```
from fastapi import FastAPI
from pydantic import BaseModel

# Creating the FastAPI app
app = FastAPI()

# Defining a Pydantic model for the query request
class QueryRequest(BaseModel):
    question: str

import nest_asyncio
import threading
import uvicorn
import logging
from fastapi import FastAPI
from pydantic import BaseModel
from transformers import pipeline
from langchain.vectorstores import FAISS
from langchain_huggingface import HuggingFaceEmbeddings  # Updated import

# Applying nest_asyncio to allow FastAPI to run on Jupyter
nest_asyncio.apply()

# Suppress all logs at the INFO level and below for relevant loggers
logging.basicConfig(level=logging.WARNING)

# Suppress Uvicorn logs
logging.getLogger("uvicorn").setLevel(logging.WARNING)
logging.getLogger("uvicorn.error").setLevel(logging.WARNING)
logging.getLogger("uvicorn.access").setLevel(logging.WARNING)

# Suppress any other third-party library logs (e.g., FAISS, Hugging Face
transformers)
logging.getLogger("transformers").setLevel(logging.WARNING)  # Example for
Hugging Face
logging.getLogger("faiss").setLevel(logging.WARNING)  # Example for FAISS

# Initializing the Hugging Face QA pipeline with a fine-tuned model
qa_pipeline = pipeline("question-answering", model="deepset/roberta-base-
squad2", tokenizer="deepset/roberta-base-squad2")

# Creating an embedding object using HuggingFaceEmbeddings
```

```
embedding_model = HuggingFaceEmbeddings(model_name="all-MiniLM-L6-v2")

# Loading the FAISS vector store (Make sure the path is correct where your
index is saved)
vector_store = FAISS.load_local("./faiss_index", embedding_model,
allow_dangerous_deserialization=True)

# Creating FastAPI app
app = FastAPI()

# Defining a Pydantic model for the query request
class QueryRequest(BaseModel):
    question: str

# Defining the FastAPI endpoint for question answering
@app.post("/ask-question/")
async def ask_question(query_request: QueryRequest):
    query = query_request.question

    # Performing similarity search on the FAISS vector store
    search_results = vector_store.similarity_search(query, k=1)

    # Only taking the top chunk as context
    context = search_results[0].page_content

    # Using the QA pipeline to generate an answer based on the context
    answer = qa_pipeline(question=query, context=context)

    # Returning the answer in the response
    return {"answer": answer['answer']}


# Function to run the FastAPI app in a separate thread
def run_fastapi():
    uvicorn.run(app, host="127.0.0.1", port=8007, log_level="info")

# Running the FastAPI app in a separate thread
thread = threading.Thread(target=run_fastapi)
thread.start()
```

```
Device set to use mps:0
INFO:     Started server process [1417]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     Uvicorn running on http://127.0.0.1:8007 (Press CTRL+C to quit)
```

```
# Creating and managing vector stores for fast similarity search
from langchain.vectorstores import FAISS
from langchain.document_loaders import PyPDFLoader
from langchain.embeddings import HuggingFaceEmbeddings
from langchain.text_splitter import RecursiveCharacterTextSplitter
```

```
# Loading documents and create embeddings
pdf_paths = ["/Users/diboshbaruah/Desktop/Capstone_Project - Generative
AI/financial risk.pdf",
             "/Users/diboshbaruah/Desktop/Capstone_Project - Generative
AI/operations risk.pdf"]
documents = []

for path in pdf_paths:
    loader = PyPDFLoader(path)
    documents.extend(loader.load())

# Splitting documents into smaller chunks for better similarity matching
text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000,
chunk_overlap=100)
chunks = text_splitter.split_documents(documents)

# Using HuggingFace embeddings
embedding_model = HuggingFaceEmbeddings(model_name="all-MiniLM-L6-v2")

# Creating the FAISS vector store
vector_store = FAISS.from_documents(chunks, embedding_model)

# Saving the FAISS index to disk
vector_store.save_local("./faiss_index")

# Sending HTTP POST requests to the FastAPI server

import requests

# The URL of the FastAPI server
url = "http://127.0.0.1:8007/ask-question/"

# Defining the question payload
question_payload = {
    "question": "How loans are processed?"
}

# Sending a POST request to the FastAPI server
response = requests.post(url, json=question_payload)

print()

# Checking the response status code and output the answer
if response.status_code == 200:
    answer = response.json()
    print(f"Answer: {answer['answer']}")
else:
    print(f"Error: {response.status_code}")
```

```
INFO:     127.0.0.1:49557 - "POST /ask-question/ HTTP/1.1" 200 OK

Answer: Loan Review Mechanism
```

** Now running the saved scripts on jupyter notebook - embedding.py // app.py // client.py ***

```
# main.py

# Running embedding.py
%run embedding.py

# Running app.py to start the FastAPI server
%run app.py

# Running client.py to test the FastAPI API call
%run client.py

Embedding and FAISS index creation completed.

Device set to use mps:0

FastAPI server started.

INFO:     Started server process [1417]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
ERROR:    [Errno 48] error while attempting to bind on address ('127.0.0.1',
8007): address already in use
INFO:     Waiting for application shutdown.
INFO:     Application shutdown complete.

INFO:     127.0.0.1:49791 - "POST /ask-question/ HTTP/1.1" 200 OK

Answer: Loan Review Mechanism
```