RAG BOT Report

Akhil Muraleedharan

January 2025

1 Introduction

This is a Retrieval-Augmented Generation (RAG) Model for QA Bot on P&L Data where we provide the bot with questions related to an uploaded document and the answers are presented by the bot.

2 Model Architecture

PypdfLoader is used to load the file so that it can be analysed by the bot. Here we will use the Sample_Financial_Statement.pdf but for the web application which will be built later, we will allow the user to upload the document. An interesting note is that the PypdfLoader loads the pdf file as pages and not as a complete document. We can access the specific page of the document by indexing the page within []. For example if we want to access the page 6, we will print it as print(data[6]), where loader = PyPDFLoader ('Sample_Financial_Statement.pdf') and data = loader.load().

We need to split the file into smaller chunks for the document to be analyzed. For this we will use the RecursiveCharacterTextSplitter class from langchain.text_splitter module which will break down the document into readable chunks for better understanding. The RecursiveCharacterTextSplitter is quite versatile and therefore is used in this particular case.

Next we need to import the GoogleGenerativeAIEmbeddings to convert the text into vector format which is how the Gemini API will access the records and provide a response. We will also require a database to store the vectors and for that we will import Chroma to store the numerical tokens. We will also require an API key in order to access the Gemini API which will be used for this code. In order to access the key, we will need load_dotenv and the API key is stored in the .env file

After storing the vectorized tokens, we will use the invoke command to call the similar values from the document and the number of instances which will be called will be determined by the search_kwargs = {"k":10} parameter. We will import the ChatGoogleGenerativeAI from langchain_google_genai which will be our large languange model which will help us chat and find answers from the document.We are using the "gemini-1.5-pro" model as our llm and by setting the temperature to 0.3, we will get less random outputs from the llm and by limiting the max tokens to 500, we will decide to limit the output that we get from the llm to 500 characters.Next we decide a system prompt which will inform the bot how to respond to queries from the user. The ChatPromptTemplate will include the format in which the message is relayed to the llm in order to elicit an answer.

Them we create a chain of events which will decide order in which the events will happen for the llm. In this particular case, the llm will get the prompt from the user or human and using the retriver, we will extract an answer from the llm with respect to the query supplied from the human counterpart. The prompt which was defined in the code informs the llm about the format of the response as well.

3 Challenges and Solutions

1. Incoherent Responses: The bot seems to provide incoherent responses when faced with similar words from the document. For example, the word income does not provide a coherent response from the bot but the word comprehensive income gives out the required response. The chunk size reduction will result in a timeout and hence could not be explored further.

2. Database Visibility: Uploading new files in the app caused an error message saying that the database is not visible. To circumvent this issue, we then added a persist directory which is added to the vector creation statement.

3. Gemini Resourse Exhausted: While deploying the app on streamlit, there was a resource exhausted code from google api and therefore the api model was converted from gemini pro to gemini flash.