RAG System Specialized for Tabular Documents

Hongxiao Chen
hc39@illinois.edu
University of Illinois Urbana-Champaign
Illinois, USA

Abstract

More and more organizations store tables placed inside the Word and PDF documents. Nevertheless, non-technical users frequently do not have the necessary expertise to question these tables directly, and large language models, in isolation, may produce incorrect values when parsing tabular data directly. We present a retrieval-augmented generation system, specialized in extracting and answering queries over tabular documents. Our pipeline takes raw documents and converts them into structured table entries, annotates each table entry with its row headers, and contextual tags and encodes using an embedding model. We index the resultant vectors in an FAISS database make it possible to perform efficient nearest-neighbor retrieval of the most relevant table segments for any natural language query. Then, an LLM generates answers by referencing retrieved table snippets to form its answers. This RAG system might increase the information retrieval in the production. The performances of LLMs acting as RAG agents are also compared in this paper.

1 Introduction

In the Information Age, organizations gather huge volumes of information in the manner of tables contained within Word and PDF documents. Navigating these tabular documents to obtain the correct answers is challenging to most users since the use of database query languages requires experts to deal with them. Large Language Models (LLMs) can relate to natural language questions but they are inclined to give inaccurate or "hallucinated" answers when relying exclusively on their own knowledge source. To solve this, we deploy a Retrieval-Augmented Generation (RAG) setup whereby we integrate the LLM with an external knowledge retrieval functionality. The objective of this project is to develop a software system which will specialize in taking table data from documents, and, after that, give back responses to users queries regarding the data extracted. With the help of retrieved factual table data, the system enriches the LLM; therefore, the answers become more grounded and factual.

2 Motivation

Non-technical employees like advertising, or marketing staff, would benefit greatly from easy access to business metrics such as those stored in tables but lack the necessary technical skills to query large databases. Or sometimes the report is provided by another company, and analysts cannot access the original data. Offering these users with a natural language interface will significantly reduces the cost of entry into data-driven decisions. Answering questions from table data is, however, a technical problem. LLMs are inclined to produce sensible sounding but wrong answers if they fail to check facts against actual data(which is called hallucination). Using a RAG approach, the system searches for and retrieves pertinent table fragments and fuses them into the prompt, thereby reducing hallucinations by anchoring the answer to actual data.

3 Intended Users

The system is designed for company staffs who manage or use a large document repository. Typical users include business analysts, marketers, and other non-technical personnel who have domain questions but no professional CS skills. For instance, marketing analysts would want sales trends or customer metrics from internal reports but may not know SQL well. Or they may find it time consuming to search in countless documents named by different years and months that are provided by other companies using different formats. These users can ask natural language questions (for example, "Which region had the most sales in Q3?") and get responses that are collected from underlying table data as well as links to the source document(s). This may assist the user in immediately realizing whether that source is exactly what they need without opening documents and reading each line.

4 System Architecture and Components

Before introducing the core steps, the author will first review three classes used in this project.

- **File**: envelops a source document, while holding its unique identifier, path to a file, and name of the file.
- **Object**: represents a content unit that was obtained from a given document, a table or a text paragraph. Each `Object` contains the ID of the originating file, its position in the document, the raw content (nested dictionary for tables or string for text), the surrounding context strings (`above` and `below`), the title and any date or keyword tags. These items make up the raw materials that are embedded and indexed.
- **Tags**: manages a global dictionary of label to object mappings, including keyword tags (such as "revenue", "growth") and date tags (such as "March 2025"). The `Tags` class offers functions to store new tags for a given object ID and to ask which objects are bearing a certain tag. By centralizing tag management, it allows for fast look-ups during embedding and retrieval.

The system consists of several key components that operate in sequence, as shown in the Figure 1:

- **Document Conversion and Table Tagging**
- **Embedding**
- **FAISS Storing and Retrieval**
- **Query Processing with LLM**

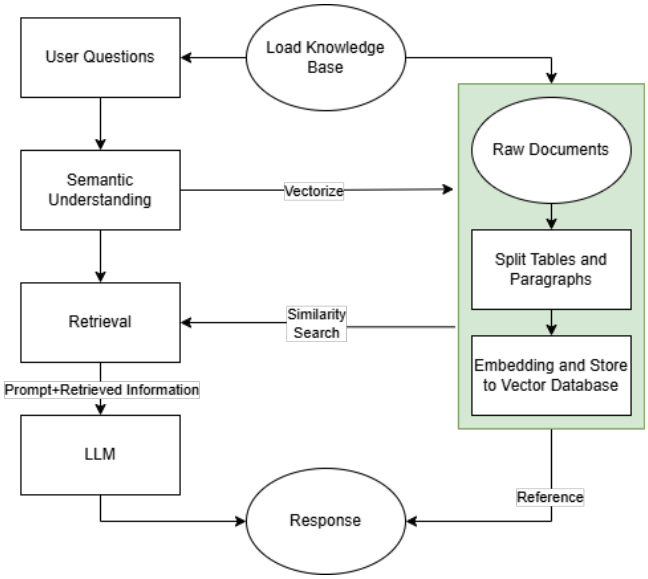

Figure 1: Key components of this RAG system

4.1 Document Conversion and Table Tagging

During the input phase, raw documents (Word or pdf) are transformed into structured documents, with the help of a script (`pdf2doc.py`). Subsequently, all Word documents are processed by a `doc2table.py` tool extracting textual paragraphs and identifying tables contained in them. The principal function of this stage is described in Algorithm 1.

Algorithm 1 Document to Table Conversion

```
function DOC2TABLE(file, tags)
    Open .docx using python-docx
    Initialize empty lists text_objects, table_objects
    for each block in .docx do
        if block is Paragraph and not empty then
            Create text Object with position index
            Append to text_objects
        else if block is Table then
            Extract title via find_title
            Break table into row-lists
            for each subtable with > 1 rows do
                Convert to DataFrame, then to dict
                Create table Object with position index
                Append to table_objects
            end for
        end if
    end for
    Call add_tags_and_contexts on table_objects, text_objects
    return (table_objects, text_objects)
end function
```

The function `doc2table` takes a File object representing a document and produces two lists of `Object` instances: one for each extracted table and one for each meaningful text paragraph. First it opens the file with `python-docx` and iterates through its body elements—distinguishing between paragraphs and tables. For each non-empty paragraph, it creates a text `Object` capturing its content, file ID, filename, and position. For each table block, it calls the `table_reader` to split and clean the raw rows, then converts each subtable into a Pandas DataFrame, and finally into a nested dictionary. Each of these dictionaries is wrapped in a table `Object` with metadata like position, title, and file info. Once all elements are collected, the function delegates to `add_tags_and_contexts` to enrich them with tags and contextual parts, which automatically detect dates from content and tile, and also search for tags that are predefined. Finally it returns the lists of fully initialized table and text `Object` ready for embedding.

4.2 Embedding

In the embedding stage, we turn each extracted Object into a set of high-dimensional vectors using `bge-base-en` embedding model[4]. The `objects_embedding` function(as shown in Algorithm 2) iterates over all objects, and for each one, constructs a small list of strings to embed, including the file name, the paragraph above and below, and a title if it exists. For table objects, we also concatenate any date tags and content tags (separated by a unique `[SEP]` token) so the model captures both when and what metadata is associated with that table entry. For text objects, we simply include the text content itself. We then call `model.embed_documents()`, which returns a list of vectors in the same order as our inputs. We map these vectors into the corresponding fields of a new `EmbeddedObject`. These embedded objects feed directly into our FAISS index for efficient similarity search during query time.

Algorithm 2 Objects Embedding

```
function OBJECTS_EMBEDDING(model, objects)
    Initialize empty list embed_objects
    for each object in objects do
        if object.content is a table (dict) then
            input_texts ← [ object.file_name, object.above, object.below, object.title, join(object.date) + "[SEP]" + object.title + "[SEP]" + join(object.tags) ]
            embedded_vectors ← model.embed_documents(input_texts)
            Create embed_object
        else
            input_texts ← [ object.file_name, object.above, object.below, object.title, object.content ]
            embedded_vectors ← model.embed_documents(input_texts)
            Create embed_object
        end if
        Append embed_object to embed_objects
    end for
    return embed_objects
end function
```

4.3 FAISS

This `FaissIdx` class, as shown in Algorithm 3, includes everything we need to build and query a vector similarity index using FAISS. In the constructor, we initialize a flat L2 index with a fixed vector dimension, set up an empty `doc_map` to keep track of which content corresponds to each vector, and store our embedding model reference. The `add_emb_doc` method accepts a pre-computed embedding vector and adds it to the index, then record the original text in `doc_map` under the next available integer counter. Finally, `search_doc` lets us perform similarity search. It embeds the user's query, searches the FAISS index for the `top-k` nearest vectors, and then looks up the corresponding texts in `doc_map`. The method returns a list of mappings from each retrieved text to its distance score, which we can then pass on to our LLM for answer generation.

Algorithm 3 FaissIdx Class

function FaissIdx.__init__(model, dim=768)
    Create FAISS L2 index with dimension dim
    Initialize doc_map dictionary
    Store embedding `model`
    Initialize counter `ctr` ← 0
end function
function FaissIdx.add_emb_doc(embedding, text)
    Reshape `embedding` to (1, dim)
    Add vector to FAISS index
    `doc_map[ctr]` ← `text`
    `ctr` ← `ctr` + 1
end function
function FaissIdx.search_doc(query, k=3)
    `query_emb` ← `model.embed_query(query)`
    Reshape `query_emb` to (1, dim)
    Search index for `k` neighbors → `D, I`
    Initialize `results` list
    for each (`index`, `distance`) in (`I[0]`, `D[0]`) **do**
        `text` ← `doc_map[index]`
        Add {`text`: `distance`} to `results`
    end for
    return `results`
end function

4.4 Query Processing with LLM

When the user submits a question through the `Gradio` frontend, the predict function ties together our FAISS retrieval and LLM answer generation, as shown in the following Python code block. First, it calls our `faiss_retriever` to fetch the top-k most relevant table or text snippets based on the user's message. Next, it constructs an LLM prompt in two parts. The first part, `query_text`, introduces the task: 'Please read the following documents.' We then append a numbered list of each retrieved snippet under clear separators. After listing the snippets, we include the user's original question with: 'Answer this according to the documents.' This tells the LLM which pieces of external data to use when crafting. Then its calls the LLM's `.answer()` method with `startup_prompt`, telling the LLM that it is professional in information retrieval and it should only answer based on provided data. The LLM returns its generated answer, which we add to the chat history alongside the original message. Finally, the function returns four items as expected by the `Gradio` interface: the updated chat history to display, the state object for future turns, a blank string to clear the input box, and the raw FAISS search results for reference.

```
def predict(message, top_k, history):
    if history is None:
        history = []
    top_res = faiss_retriever.search_doc(message, k=top_k)

    query = "Please read the following documents\n"
    search_res = ""
    for i in range(len(top_res)):
        search_res += f"No.{i+1} Search Result:\n"
        search_res += next(iter(top_res[i])) + "\n"

    query += "Answer this according to the documents"
             + message + "\n"

    if llm is None:
        answer = "LLM not loaded, returning FAISS results"
    else:
        logger.info("To LLM: " + query + search_res)
        answer, _ = llm.answer(query + search_res,
                               startup_prompt)
    history.append((message, answer))
    return history, history, "", search_res
```

5 User Guide

5.1 Setup

After downloading it from GitHub, users should also download BGE model and place it in `src\models\baai\bge-base-en-v1.5` or modify `ui.py` to change the model name into `BAAI/bge-base-en-v1.5` to download it automatically. If users wish to use to locally deployed LLM, they should also do the same for LLM part in `ui.py`. Otherwise, they can create a `config.json` under the root directory and add their API keys into it, like this:

```
{
  "ChatGLM_api_key": "xxx"
}
```

5.2 Import Documents

Before using this system, users should place the documents under the folder with the knowledge base name in `.\data\inputdata`. For example, if the knowledge base name is `Store-Sales`, make sure the documents for this knowledge base is located in `.\data\inputdata\Store-Sales`.

The accepted formats are: `.docx`, `.doc`, `.pdf`.

5.3 RAG Frontend

The user interface of this RAG system is built by Gradio[1], which is shown in Figure 2. First, users can select their preferred embedding model and LLM on the top left part. They can adjust the `Top-K documents` slider to control how many candidate snippets they want to retrieve from FAISS. Higher `Top-K` will increase the chance

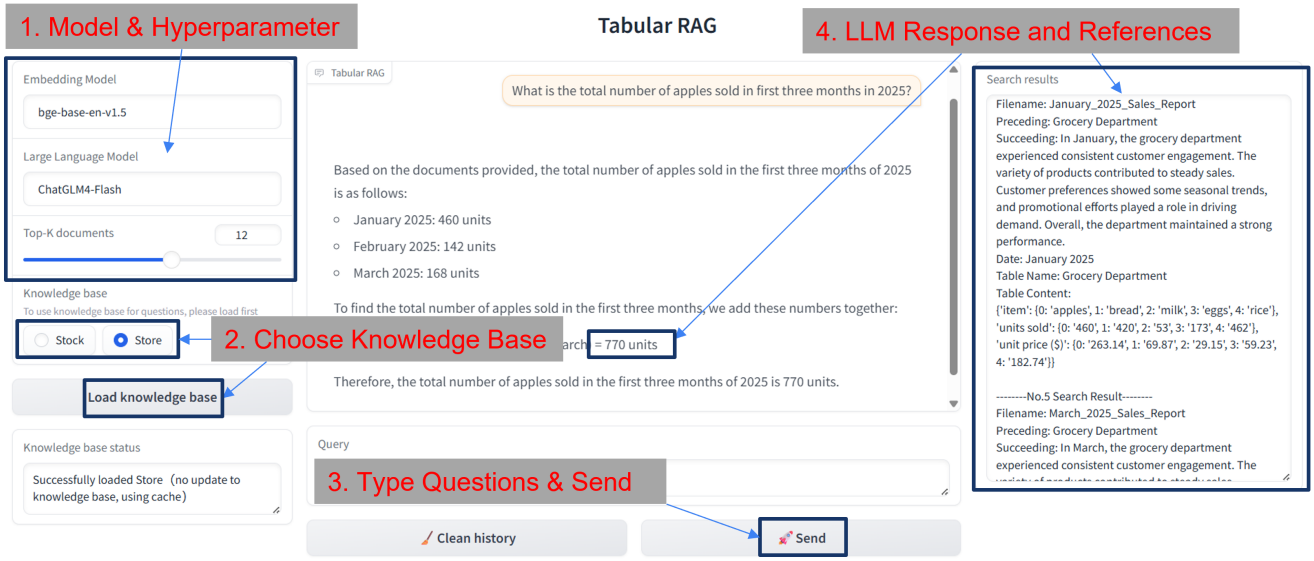

Figure 2: Gradio UI

of retrieving the desired information, but will result in longer waiting time. Before asking questions, user should use buttons(on the lower left part) for each knowledge base to pick the folder containing the documents they want to query. After clicking `Load knowledge base`, the status box below will confirm if the loading process has completed (or that it is using cached data). The user can then type their natural language questions into the query box and hit the `Send` button. They can clear the history with the `Clean history` button. Finally, the central chat window displays the final answer synthesized by the LLM. On the right are the raw FAISS search results displayed: each snippet is labeled with its source file and any metadata of it. This feature allows users to verify exactly where each piece of information came from.

6 Performance Comparison of LLM Models

We evaluated three LLMs—ChatGLM-4-Flash[2], Gemini-2.0-Flash[3], and GLM-Z1-Flash[2]—on two classes of tasks: basic information retrieval (IR) from tables with some easy computations(like addition from multiple tables) and tasks involves complex computation(like multiplication and division). All models used identical retrieved data(with `top-k`= 12) and prompt templates. The test documents are AI-generated sales reports of a store in first three months of a year.

For the metrics:

$$\text{IR Accuracy} = \frac{\text{Correct LLM Responses}}{\text{All LLM Responses}}$$

$$\text{Computation Accuracy} = \frac{\text{Correct Computation Steps Involved}}{\text{All Computation Steps Involved}}$$

6.1 Basic Information Retrieval

One example of such questions is: "What is the total number of apples sold in the first month of 2025?".

All three models demonstrated great performance when asked to extract factual values directly from retrieved table snippets with little computation needed, indicating that retrieval grounding was effective across models.

Table 1: Basic IR Accuracy by Model

Model	IR Accuracy (%)
ChatGLM-4-Flash	100.0
Gemini-2.0-Flash	100.0
GLM-Z1-Flash	100.0

6.2 Complex Information Retrieval

One example of such questions is: "What is the total revenue of selling apples in first three months of 2025?".

When the query required the LLM to perform complex numeric computations over table values (e.g. unit price∗items sold, $\frac{\texttt{Janurary items sold}}{\texttt{All items sold}}$), ChatGLM-4-Flash exhibited significantly lower accuracy than the other two models. The reasoning model GLM-Z1-Flash has the best performance, but with worst response speed(very small difference compared to other models, acceptable).

The author noticed that although these LLMs are equally perfect at addition, ChatGLM-4-Flash almost failed all float multiplications, Gemini-2.0-Flash sometimes made errors, and GLM-Z1-Flash producing almost no errors.

Table 2: Complex IR Accuracy by Model

Model	IR Accuracy (%)	Comput. Accuracy (%)
ChatGLM-4-Flash	20.0	29.4
Gemini-2.0-Flash	70.0	85.2
GLM-Z1-Flash	90.0	97.0

The results indicate that all models are good at basic IR and simple additions. ChatGLM-4-Flash has difficulties with float multiplications and should be avoided for complex tasks. On the other hand, both Gemini-2.0-Flash and GLM-Z1-Flash produce high computation accuracy, thus making them desirable when it comes to queries that require any complex numerical computation. Since Gemini is not available in some regions, the author suggests using GLM-Z1-Flash due to the fact that it is one of the few free reasoning models available for small business and personal use.

7 Summary

To conclude, this project started with the challenge that business tables hide crucial insights but are locked behind formats and technical barriers. LLMs can't reliably read raw tables without hallucinating. The solution is a Retrieval-Augmented Generation system specifically designed for tabular documents. This system convert documents into structured objects, enrich them with header, context, and tags, then embed them using a BGE model. FAISS allows fast retrieval, and the final answer is created by an LLM, always giving the source file as reference. This method gives non-technical employees the ability to perform natural language queries and stops LLMs from hallucinating by linking data with questions. The author suggests using Gemini for fast respond and GLM-Z1-Flash for complex tasks. In the future, the author will try using Chain of Thoughts to provide examples of how to do calculations for common equations in data analysis. The author will also introduce a relation database like Neo4j to use relationships as tags for better embedding of tables with professional terminologies.

References

[1] Abubakar Abid, Ali Abdalla, Ali Abid, Dawood Khan, Abdulrahman Alfozan, and James Zou. 2019. Gradio: Hassle-Free Sharing and Testing of ML Models in the Wild. *arXiv preprint arXiv:1906.02569* (2019).

[2] Team GLM, Aohan Zeng, Bin Xu, Bowen Wang, Chenhui Zhang, Da Yin, Diego Rojas, Guanyu Feng, Hanlin Zhao, Hanyu Lai, Hao Yu, Hongning Wang, Jiadai Sun, Jiajie Zhang, Jiale Cheng, Jiayi Gui, Jie Tang, Jing Zhang, Juanzi Li, Lei Zhao, Lindong Wu, Lucen Zhong, Mingdao Liu, Minlie Huang, Peng Zhang, Qinkai Zheng, Rui Lu, Shuaiqi Duan, Shudan Zhang, Shulin Cao, Shuxun Yang, Weng Lam Tam, Wenyi Zhao, Xiao Liu, Xiao Xia, Xiaohan Zhang, Xiaotao Gu, Xin Lv, Xinghan Liu, Xinyi Liu, Xinyue Yang, Xixuan Song, Xunkai Zhang, Yifan An, Yifan Xu, Yilin Niu, Yuantao Yang, Yueyan Li, Yushi Bai, Yuxiao Dong, Zehan Qi, Zhaoyu Wang, Zhen Yang, Zhengxiao Du, Zhenyu Hou, and Zihan Wang. 2024. ChatGLM: A Family of Large Language Models from GLM-130B to GLM-4 All Tools. arXiv:2406.12793 [id='cs.CL']

[3] Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, et al. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805* (2023).

[4] Peitian Zhang, Shitao Xiao, Zheng Liu, Zhicheng Dou, and Jian-Yun Nie. 2023. Retrieve Anything To Augment Large Language Models. arXiv:2310.07554 [cs.IR]