0.1 Semi-structured RAG

Blog post

Many documents contain a mixture of content types, including text and tables.

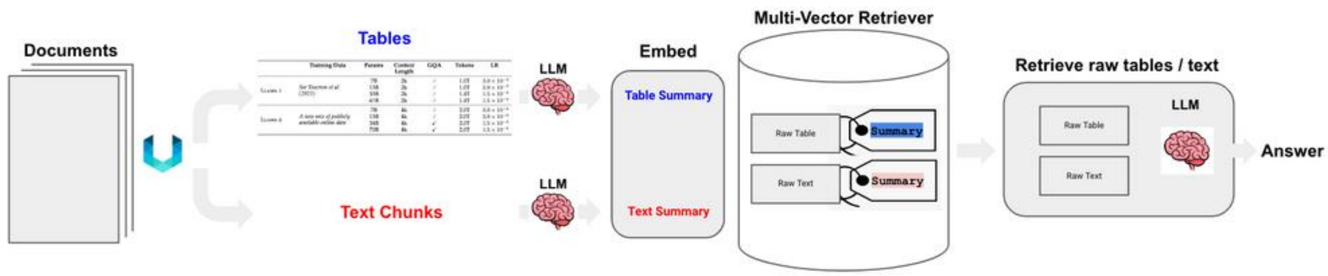
Semi-structured data can be challenging for conventional RAG for at least two reasons:

- Text splitting may break up tables, corrupting the data in retrieval
- Embedding tables may pose challenges for semantic similarity search

This cookbook shows how to perform RAG on documents with semi-structured data:

- We will use Unstructured to parse both text and tables from documents (PDFs).
- We will use the multi-vector retriever to store raw tables, text along with table summaries better suited for retrieval.
- We will use LCEL to implement the chains used.

The overall flow is here:

- 

0.2 Packages

```
[ ]: %%capture
! pip install langchain unstructured[all-docs] pydantic lxml langchainhub fastapi kaleido uvicorn
```

The PDF partitioning used by Unstructured will use:

- `tesseract` for Optical Character Recognition (OCR)
- `poppler` for PDF rendering and processing

```
[ ]: #! brew install tesseract
#! brew install poppler
```

```
[ ]: %%capture
!sudo apt-get install poppler-utils tesseract-ocr
```

0.3 Data Loading

0.3.1 Partition PDF tables and text

Apply to the LLaMA2 paper.

We use the Unstructured partition_pdf, which segments a PDF document by using a layout model.

This layout model makes it possible to extract elements, such as tables, from pdfs.

We also can use Unstructured chunking, which:

- Tries to identify document sections (e.g., Introduction, etc)
- Then, builds text blocks that maintain sections while also honoring user-defined chunk sizes

[ ]:
```
import urllib.request

url = "https://arxiv.org/pdf/2307.09288.pdf"
filename = "Llama2.pdf"
urllib.request.urlretrieve(url, filename)
```

[ ]:
```
('Llama2.pdf', <http.client.HTTPMessage at 0x7fa585f655a0>)
```

[ ]:
```
path = "/content/"
```

[ ]:
```
from lxml import html
from pydantic import BaseModel
from typing import Any, Optional
from unstructured.partition.pdf import partition_pdf

# Get elements
raw_pdf_elements = partition_pdf(filename=path+"Llama2.pdf",
                                 # Unstructured first finds embedded image␣
 ↪blocks
                                 extract_images_in_pdf=False,
                                 # Use layout model (YOLOX) to get bounding␣
 ↪boxes (for tables) and find titles
                                 # Titles are any sub-section of the document
                                 infer_table_structure=True,
                                 # Post processing to aggregate text once we␣
 ↪have the title
                                 chunking_strategy="by_title",
                                 # Chunking params to aggregate text blocks
                                 # Attempt to create a new chunk 3800 chars
                                 # Attempt to keep chunks > 2000 chars
                                 max_characters=4000,
                                 new_after_n_chars=3800,
                                 combine_text_under_n_chars=2000,
                                 #image_output_dir_path=path
```

```
)
```

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt.zip.
[nltk_data] Downloading package averaged_perceptron_tagger to
[nltk_data]     /root/nltk_data...
[nltk_data]   Unzipping taggers/averaged_perceptron_tagger.zip.

Downloading (...)l0.05_quantized.onnx:   0%|          | 0.00/54.6M [00:00<?, ?B/s]

Downloading (...)lve/main/config.json:   0%|          | 0.00/1.47k [00:00<?, ?B/s]

Downloading model.safetensors:   0%|          | 0.00/115M [00:00<?, ?B/s]

Downloading model.safetensors:   0%|          | 0.00/46.8M [00:00<?, ?B/s]

Some weights of the model checkpoint at microsoft/table-transformer-structure-recognition were not used when initializing TableTransformerForObjectDetection:
['model.backbone.conv_encoder.model.layer3.0.downsample.1.num_batches_tracked',
'model.backbone.conv_encoder.model.layer2.0.downsample.1.num_batches_tracked',
'model.backbone.conv_encoder.model.layer4.0.downsample.1.num_batches_tracked']
- This IS expected if you are initializing TableTransformerForObjectDetection
from the checkpoint of a model trained on another task or with another
architecture (e.g. initializing a BertForSequenceClassification model from a
BertForPreTraining model).
- This IS NOT expected if you are initializing
TableTransformerForObjectDetection from the checkpoint of a model that you
expect to be exactly identical (initializing a BertForSequenceClassification
model from a BertForSequenceClassification model).
```

We can examine the elements extracted by `partition_pdf`.

`CompositeElement` are aggregated chunks.

[ ]:
```
# Create a dictionary to store counts of each type
category_counts = {}

for element in raw_pdf_elements:
    category = str(type(element))
    if category in category_counts:
        category_counts[category] += 1
    else:
        category_counts[category] = 1

# Unique_categories will have unique elements
unique_categories = set(category_counts.keys())
category_counts
```

[ ]:
```
{"<class 'unstructured.documents.elements.CompositeElement'>": 120,
 "<class 'unstructured.documents.elements.Table'>": 47,
 "<class 'unstructured.documents.elements.TableChunk'>": 2}
```

[ ]:
```python
class Element(BaseModel):
    type: str
    text: Any

# Categorize by type
categorized_elements = []
for element in raw_pdf_elements:
    if "unstructured.documents.elements.Table" in str(type(element)):
        categorized_elements.append(Element(type="table", text=str(element)))
    elif "unstructured.documents.elements.CompositeElement" in
 ↪str(type(element)):
        categorized_elements.append(Element(type="text", text=str(element)))

# Tables
table_elements = [e for e in categorized_elements if e.type == "table"]
print(len(table_elements))

# Text
text_elements = [e for e in categorized_elements if e.type == "text"]
print(len(text_elements))
```

```
49
120
```

0.4 Multi-vector retriever

Use multi-vector-retriever to produce summaries of tables and, optionally, text.

With the summary, we will also store the raw table elements.

The summaries are used to improve the quality of retrieval, as explained in the multi vector retriever docs.

The raw tables are passed to the LLM, providing the full table context for the LLM to generate the answer.

0.4.1 Summaries

[ ]:
```python
from langchain.chat_models import ChatOpenAI
from langchain.prompts import ChatPromptTemplate
from langchain.schema.output_parser import StrOutputParser
```

We create a simple summarize chain for each element.

You can also see, re-use, or modify the prompt in the Hub here.

```
from langchain import hub
obj = hub.pull("rlm/multi-vector-retriever-summarization")
```

[ ]:
```
%%capture
!pip install openai
```

[ ]:
```
import openai
import os

# find API key in console at https://platform.openai.com/account/api-keys

os.environ["OPENAI_API_KEY"] = "YOUR_OPENAI_API_KEY"
openai.api_key = os.environ["OPENAI_API_KEY"]
```

[ ]:
```
os.environ["LANGCHAIN_TRACING_V2"]="true"
os.environ["LANGCHAIN_ENDPOINT"]="https://api.smith.langchain.com"
os.environ["LANGCHAIN_API_KEY"]="YOUR_LANGCHAIN_API_KEY"
os.environ["LANGCHAIN_PROJECT"]="langchain_semi_structured_RAG"
```

[ ]:
```
# Prompt
prompt_text="""You are an assistant tasked with summarizing tables and text. \
Give a concise summary of the table or text. Table or text chunk: {element} """
prompt = ChatPromptTemplate.from_template(prompt_text)

# Summary chain
model = ChatOpenAI(temperature=0,model="gpt-4")
#model = ChatOpenAI(temperature=0,model="gpt-3.5-turbo")

summarize_chain = {"element": lambda x:x} | prompt | model | StrOutputParser()
```

[ ]:
```
# Apply to tables
tables = [i.text for i in table_elements]
table_summaries = summarize_chain.batch(tables, {"max_concurrency": 5})
```

[ ]:
```
# Apply to texts
texts = [i.text for i in text_elements]
text_summaries = summarize_chain.batch(texts, {"max_concurrency": 5})
```

0.4.2 Add to vectorstore

Use Multi Vector Retriever with summaries:

- InMemoryStore stores the raw text, tables
- vectorstore stores the embedded summaries

[ ]:
```
%%capture
!pip install chromadb tiktoken
```

[ ]:
```
import uuid
from langchain.vectorstores import Chroma
from langchain.storage import InMemoryStore
```

```python
from langchain.schema.document import Document
from langchain.embeddings import OpenAIEmbeddings
from langchain.retrievers.multi_vector import MultiVectorRetriever

# The vectorstore to use to index the child chunks
vectorstore = Chroma(
    collection_name="summaries",
    embedding_function=OpenAIEmbeddings()
)

# The storage layer for the parent documents
store = InMemoryStore()
id_key = "doc_id"

  # The retriever (empty to start)
retriever = MultiVectorRetriever(
    vectorstore=vectorstore,
    docstore=store,
     id_key=id_key,
)

# Add texts
doc_ids = [str(uuid.uuid4()) for _ in texts]
summary_texts = [Document(page_content=s,metadata={id_key: doc_ids[i]}) for i,
 ↪s in enumerate(text_summaries)]
retriever.vectorstore.add_documents(summary_texts)
retriever.docstore.mset(list(zip(doc_ids, texts)))

# Add tables
table_ids = [str(uuid.uuid4()) for _ in tables]
summary_tables = [Document(page_content=s,metadata={id_key: table_ids[i]}) for
 ↪i, s in enumerate(table_summaries)]
retriever.vectorstore.add_documents(summary_tables)
retriever.docstore.mset(list(zip(table_ids, tables)))
```

0.5 RAG from LangChain Expression Language.

Run RAG pipeline.

[ ]:
```python
from operator import itemgetter
from langchain.schema.runnable import RunnablePassthrough

# Prompt template
template = """Answer the question based only on the following context, which
 ↪can include text and tables:
{context}
Question: {question}
```

```
"""
prompt = ChatPromptTemplate.from_template(template)

# LLM
model = ChatOpenAI(temperature=0,model="gpt-4")
#model = ChatOpenAI(temperature=0,model="gpt-3.5-turbo")

# RAG pipeline
chain = (
    {"context": retriever, "question": RunnablePassthrough()}
    | prompt
    | model
    | StrOutputParser()
)
```

[ ]:
```
chain.invoke("What is the number of training tokens for LLaMA2?")
```

[ ]: 'LLaMA 2 was pretrained on 2 trillion tokens of data from publicly available sources.'

[ ]:
```
chain.invoke("What is the average tokens in prompt for Stanford SHP?")
```

[ ]: 'The average number of tokens in the prompt for Stanford SHP is 338.3.'

[ ]:
```
chain.invoke("What is the average tokens in responses for Meta?")
```

[ ]: 'The average number of tokens in responses for Meta is 234.1.'

[ ]:
```

```

[ ]: 'The average tokens in responses for Meta is 31.4.'

[ ]:
```
chain.invoke("What is the Pretraining data ?")
```

```
WARNING:urllib3.connectionpool:Retrying (Retry(total=2, connect=None, read=None,
redirect=None, status=None)) after connection broken by
'RemoteDisconnected('Remote end closed connection without response')':
/runs/9be6bdb4-0933-4777-b41a-438c4441da8d
```

[ ]: 'The pretraining data for Llama 2 models includes a new mix of data from publicly available sources, which does not include data from Meta’s products or services. Efforts were made to remove data from certain sites known to contain a high volume of personal information about private individuals. The training was performed on 2 trillion tokens of data, up-sampling the most factual sources in an effort to increase knowledge and dampen hallucinations. The pretraining data has a cutoff of September 2022.'

We can check the trace to see what chunks were retrieved.

```
[ ]:
```