Preprocessing Unstructured Data for LLM Applications

Last updated: July 10, 2024

Contents

1 Introduction **1**

2 Overview of LLM Data Preprocessing **2**
2.1 Retrieval-Augmented Generation . . . 2
2.2 Preprocessing Outputs . . . 2
2.3 Why is Data Preprocessing Hard? . . . 2

3 Normalizing the Content **3**
3.1 General Concept . . . 3
3.1.1 Normalization Benefits . . . 3
3.1.2 Data Serialization . . . 3
3.2 Preprocessing Different Document Types . . . 3
3.2.1 HTML . . . 3
3.2.2 PowerPoint . . . 3
3.2.3 PDF . . . 3
3.3 Code for Normalizing Document Content . . . 4
3.3.1 General Code . . . 4
3.3.2 HTML Documents . . . 4
3.3.3 PowerPoint Documents . . . 4
3.3.4 PDF Documents . . . 5

4 Metadata Extraction and Chunking **6**
4.1 What is Metadata? . . . 6
4.1.1 Semantic Search . . . 6
4.1.2 Semantic Search Challenges . . . 6
4.1.3 Hybrid Search . . . 6
4.2 Chunking . . . 6
4.2.1 Chunk by Elements . . . 6
4.3 Practical Metadata Extraction and Chunking . . . 7
4.3.1 General Code . . . 7
4.3.2 Run the Document through the API . . . 7
4.3.3 Find Elements Associated with Chapters . . . 8
4.3.4 Load Documents into a Vector Database . . . 8
4.3.5 Perform a Hybrid Search with Metadata . . . 9
4.3.6 Chunking Content . . . 9

5 Preprocessing PDFs and Images **10**
5.1 Document Image Analysis . . . 10
5.1.1 Document Layout Detection . . . 10
5.1.2 Vision Transformers . . . 10
5.1.3 When to Use DLD or ViT Models . . . 10
5.2 Practical Exercise . . . 11
5.2.1 General Code . . . 11
5.2.2 Preprocess as HTML . . . 11
5.2.3 Preprocess with Fast Strategy . . . 11
5.2.4 Preprocess with DLD Model . . . 11
5.2.5 Compare Outputs . . . 12

6 Extracting Tables **13**
6.1 Table Extraction . . . 13
6.1.1 Table Transformers . . . 13

6.1.2 Vision Transformers . . . 13
6.1.3 OCR Postprocessing . . . 13
6.2 Practical Exercise . . . 13
6.2.1 General Code . . . 13
6.2.2 Extracting Tables . . . 14

7 Build Your Own RAG Bot **16**
7.1 General Code . . . 16
7.2 Preprocess the PDF . . . 16
7.2.1 Filter References Section . . . 17
7.2.2 Filter out Headers . . . 17
7.3 Preprocess the PowerPoint . . . 17
7.4 Preprocess the README . . . 17
7.5 Load the Documents into the Vector DB . . . 17

1 Introduction

Retrieval-Augmented Generation (RAG) has become widely adopted in many enterprises. The typical RAG pipeline consists of several key components, including data loading, chunking, embedding, storing in the vector database, and then retrieval.

In this course, we will explore techniques to represent various types of unstructured data, such as text, images, and tables, from many different sources like PDFs, PowerPoint presentations, and Word documents. The goal is to make all this information accessible to the LLM RAG pipeline.

A particularly challenging task in RAG is data loading and chunking because data is stored in various file types and formats. Each file type, such as Excel, PDF, Markdown, PowerPoint, Keynote, Outlook, or Slack, may contain data in different formats like tables, images, simple text, or bullet lists. Therefore, a data loader must first be capable of parsing many different file formats. Once the data is parsed, it is highly beneficial to normalize it. For example, when normalizing a table from a PDF, PowerPoint, or other sources, it is useful to represent all tables in a consistent manner. It is also important to maintain some degree of the original document's structure by preserving this structured information in metadata. For instance, by recording that a paragraph has a parent, which is the title of the chapter, a query matching that chapter can be expanded to return child text as well. This organization of data in a hierarchical tree structure is extremely useful.

2 Overview of LLM Data Preprocessing

In this section, we will cover the fundamentals of data preprocessing and discuss why it is a crucial and challenging problem when building LLM applications that use data from different sources.

2.1 Retrieval-Augmented Generation

Retrieval-Augmented Generation (RAG) is a technique for grounding LLM responses on validated external information. This information can be contained in various document types, such as emails or PDFs. Conceptually, RAG applications load this external context into a database, retrieve that content, insert it into a prompt, and then pass that prompt to an LLM. This process is known as *contextual integration*, where the external information is seamlessly integrated into the prompt. By the time a question reaches the LLM, it contains relevant external information that the LLM can use to construct its response.

2.2 Preprocessing Outputs

Preprocessing documents involves several steps:

1. **Extract the document content**: This refers to extracting the text content from documents, which is used for search in RAG applications.
2. **Extract document elements**: These are the basic building blocks of the document, such as titles, narrative text, lists, tables, and images. These elements are useful for various RAG tasks such as filtering and chunking.
3. **Extract element metadata**: This includes additional information about an element, such as file name, page number, or file type. Metadata is useful for filtering in hybrid searches and for identifying the source of a response.

2.3 Why is Data Preprocessing Hard?

There are several reasons why data preprocessing is a challenging task:

- Different document types have *different content cues*. For example, an HTML file may use tag names to indicate whether a piece of text is a title or a list, while PDFs use visual cues. Preprocessing these different document types in a common manner requires understanding how each document type indicates different elements.
- Documents come in a *variety of formats*, and it is necessary to *standardize* these so that the application can process them uniformly. Standardizing is difficult because documents have different formats and structures.
- Due to this *extraction variability*, data preprocessing techniques must account for differences in how data is extracted from various document types.
- Understanding information about document *structure* is essential to *extract* meaningful *metadata*, which can be used for various operations in RAG applications.

3 Normalizing the Content

In this section, we will explore how to extract and normalize content from a diverse range of document types so that the LLM can reference information from PDFs, PowerPoint, Word documents, HTML, and more.

3.1 General Concept

Documents come in a variety of formats, and when we're building an LLM application, we don't want to worry about the source of our documents. We want them all in a common format so that our LLM application can treat them consistently.

The first step in achieving this is to **break down the documents into common elements** like titles and narrative text.

3.1.1 Normalization Benefits

There are several benefits to normalization:

- First, it allows us to *process any document in the same way* regardless of the source format. This enables operations like filtering out unwanted elements such as headers and footers, or applying downstream operations like chunking uniformly across different document types.
- Second, it *reduces processing costs*. Typically, the most expensive part of preprocessing documents is extracting the initial content. Downstream operations like chunking are generally inexpensive. By normalizing all content to the same format, we can experiment with different chunking strategies in a computationally inexpensive manner.

3.1.2 Data Serialization

Once we've normalized the content, the next step is typically data serialization. This allows the results of document preprocessing to be reused later.

In this course, we'll serialize data as *JSON*, which is convenient for several reasons:

- It's a common and well-understood structure.
- It's a standard HTTP response format, which is useful when processing documents like PDFs and images that require model-based workloads run over an API.
- It can be used in multiple programming languages.
- It can be converted to JSONL for streaming use cases.

3.2 Preprocessing Different Document Types

Now we'll learn about preprocessing a few different specific document types.

3.2.1 HTML

HTML is relevant because, in many cases, we want to read content from the internet into our LLM application. When processing HTML documents, we typically look at *HTML tags*. For instance, an `<h1>` tag will indicate that content is likely a title, while a `<p>` (paragraph) tag will indicate that content is likely narrative text.

In addition to using these tags, it's also useful to employ natural language processing capabilities to understand the content better. For example, long content with multiple sentences in a paragraph tag is likely to be narrative text, whereas short, all-caps content in a paragraph tag may be more likely to be a title.

Thus, we can use both structured and unstructured information within the document to categorize the elements.

3.2.2 PowerPoint

PowerPoint documents are widely used in various business areas such as consulting. For corporate use cases, this is crucial for expanding the LLM's knowledge of the organization.

The extraction process for Microsoft PowerPoint is very similar to HTML: under the hood, pptx files consist of *XML* that can be preprocessed using *rules-based logic*. This involves parsing elements like bulleted paragraphs, slide notes, shapes, and tables. Python libraries such as `python-pptx` make it easy to navigate and extract textual or visual information from slides.

3.2.3 PDF

PDF files often have complex layouts and tables.

PDFs differ from HTML or PowerPoint, where we rely on semi-structured information to identify element types. In PDFs, we look for *visual cues*. For example, text within a PDF that is bold and underlined may be more likely to be a title; longer, blockier text that contains multiple sentences and lacks emphasis like bolding or underlining is more likely to be narrative text. To understand how to divide elements, OCR and transformer-based technologies are used.

3.3 Code for Normalizing Document Content

Let's look at the code for handling various types of documents.

3.3.1 General Code

First, we import a library to filter out warnings:

```python
# Warning control
import warnings
warnings.filterwarnings('ignore')
```

Next, we import some helper functions, including those from the unstructured open source library:

```python
from IPython.display import JSON

import json

from unstructured_client import UnstructuredClient
from unstructured_client.models import shared
from unstructured_client.models.errors import SDKError

from unstructured.partition.html import partition_html
from unstructured.partition.pptx import partition_pptx
from unstructured.staging.base import dict_to_elements, elements_to_json
```

Now, set up what is needed to use the unstructured API:

```python
from Utils import Utils
utils = Utils()

DLAI_API_KEY = utils.get_dlai_api_key()
DLAI_API_URL = utils.get_dlai_url()

s = UnstructuredClient(
    api_key_auth=DLAI_API_KEY,
    server_url=DLAI_API_URL,
)
```

We will use the unstructured API for processing PDFs or images in this course because these are expensive model-based workloads requiring more setup. This code sets up our credentials to use the unstructured API.

3.3.2 HTML Documents

Using a document from the unstructured blog, first load the file and partition it using the unstructured library:

```python
filename = "example_files/medium_blog.html"
elements = partition_html(filename=filename)
```

Now, convert the output to a dictionary and then write it as JSON:

```python
element_dict = [el.to_dict() for el in elements]
example_output = json.dumps(element_dict[11:15], indent=2)
print(example_output)
```

This code simplifies navigating through the JSON object:

```python
JSON(example_output)
```

With these steps, we have broken down the HTML file into text blocks, indicating whether each is a title, narrative text, and so on.

3.3.3 PowerPoint Documents

First, import and partition the document, similar to the previous step:

```python
filename = "example_files/msft_openai.pptx"
elements = partition_pptx(filename=filename)
```

As before, convert this output to a dictionary and then to JSON:

```python
element_dict = [el.to_dict() for el in elements]
JSON(json.dumps(element_dict[:], indent=2))
```

This is all we need to transcribe and categorize each part of the presentation.

3.3.4 PDF Documents

Start by importing the file name; then, instead of passing it to the unstructured library, pass it to the unstructured API:

```python
filename = "example_files/CoT.pdf"

with open(filename, "rb") as f:
    files=shared.Files(
        content=f.read(),
        file_name=filename,
    )

req = shared.PartitionParameters(
    files=files,
    strategy='hi_res',
    pdf_infer_table_structure=True,
    languages=["eng"],
)
try:
    resp = s.general.partition(req)
    print(json.dumps(resp.elements[:3], indent=2))
except SDKError as e:
    print(e)
```

In this case, we use the API because PDF processing is a model-based workload, which is computationally expensive and harder to run locally.

The API call returns a JSON, which we can view and explore with the JSON display function from IPython:

```python
JSON(json.dumps(resp.elements, indent=2))
```

Despite the differences in file types, the result is the same: extracting and categorizing each part of the document (including the table).

4 Metadata Extraction and Chunking

In this section, we'll learn how to enrich extracted content with metadata, which helps improve downstream RAG results by supporting hybrid search and allowing us to chunk content more meaningfully for semantic search.

4.1 What is Metadata?

Metadata is additional information that we extract while preprocessing the document. Metadata can be at the *document level* or the *element level*, and it can be something extracted from the document information itself, like the last modified date or the file name, or something inferred during preprocessing, such as the category of the element type or hierarchical relationships between different element types; these are called **structural metadata**.

These metadata fields will be crucial when creating our RAG application, particularly for applications like hybrid search.

4.1.1 Semantic Search

Before learning about hybrid search, it is important to first understand some basics about semantic search for LLMs.

In a RAG system, the first step is typically retrieving documents from a vector database. The most basic way to do this is through semantic search, looking for *content that is similar to the query*.

After loading our documents into a vector database (by transforming the text into vectors through the embedding process), we run a query and search for documents based on their similarity score, which is measured by distance.

4.1.2 Semantic Search Challenges

There are some cases where similarity search isn't ideal for returning information for our RAG system. For instance:

- There could be *too many matches*, especially if the document is about the same topic. Searching on that topic might return excessive information.
- We may want to *bias our results based on other information*, such as more recent information, meaning we might prefer the most similar content within a certain time frame.
- We also *lose important information* when only searching on semantic similarity. There is valuable information contained within the document, such as section information, that might inform our search results.

4.1.3 Hybrid Search

Hybrid search addresses these challenges by *combining similarity search with structured information* extracted from the document as metadata. We can use metadata as filtering options.

4.2 Chunking

Metadata is useful not only for hybrid search but for other operations as well, such as chunking.

Chunking involves taking a long piece of text, like a large document, and breaking it down into smaller pieces so that we can pass those smaller pieces into the vector database. Then we can include those snippets in prompt templates to pass to an LLM.

There are two primary reasons for chunking:

- Some LLMs have a *limited context window*, meaning we can't pass the full document to the LLM, so we have to break it up.
- In many cases, LLMs *cost more* if we use a larger context window, so chunking documents into smaller pieces allows us to save money on inference costs.

Our similarity search queries will change based on how we chunk our content. Generally, better chunks result in better similarity search outputs, which in turn improve our end results when querying our LLM.

The simplest way to chunk is by even-sized chunks. Multiple methods exist: one is chunking by characters, another by tokens. The idea is to take a big document, set a threshold, and split off into a new chunk whenever we hit the threshold.

However, by using the metadata we can extract, we can chunk in a more intelligent way by using information about the document elements.

4.2.1 Chunk by Elements

Conceptually, this differs from traditional chunking techniques. Traditional chunking involves splitting a big document, while chunking from elements involves **first partitioning** the document **into atomic** document **elements** and **then combining** those elements into chunks.

First, partition the document as learned in the previous section. Once partitioned, we have individual document elements like titles, narrative text, and lists that we can use as the basis for our chunking operations.

Once we have these atomic elements, we can combine them into chunks. When combining them, apply break conditions to group content that logically belongs together.

For example, a break condition could be creating a new chunk whenever we hit a title. Titles often indicate the start of a new section, so applying a break condition like a title groups content from the same section into the same chunk.

By applying a break condition such as titles, we're more likely to keep together content that belongs to the same section, allowing us to construct more coherent chunks. This means more relevant content returns during similarity searches in our vector database. Thus, leveraging document structure information enables intelligent chunk construction.

Additionally, applying chunking to the list of standardized document elements allows us to experiment rapidly with different chunking techniques and see the results in our LLM end user app.

4.3 Practical Metadata Extraction and Chunking

Let's see how to practically extract and handle metadata.

4.3.1 General Code

First, import some helper functions:

```python
# Warning control
import warnings
warnings.filterwarnings('ignore')

import logging
logger = logging.getLogger()
logger.setLevel(logging.CRITICAL)

import json
from IPython.display import JSON

from unstructured_client import UnstructuredClient
from unstructured_client.models import shared
from unstructured_client.models.errors import SDKError

from unstructured.chunking.basic import chunk_elements
from unstructured.chunking.title import chunk_by_title
from unstructured.staging.base import dict_to_elements

import chromadb
```

We've also imported a new library: *ChromaDB*, an in-memory vector database for conducting similarity searches.

Next, set up the unstructured API connection:

```python
from Utils import Utils
utils = Utils()

DLAI_API_KEY = utils.get_dlai_api_key()
DLAI_API_URL = utils.get_dlai_url()

s = UnstructuredClient(
    api_key_auth=DLAI_API_KEY,
    server_url=DLAI_API_URL,
)
```

In this example, we'll work with an EPUB document about winter sports in Switzerland. The goal is to identify the chapter for each section and conduct a hybrid search where we'll ask a question about a particular chapter.

4.3.2 Run the Document through the API

The first step is to run the document through the unstructured API. We use the API because EPUB files are converted to HTML before preprocessing.

```python
filename = "example_files/winter-sports.epub"

with open(filename, "rb") as f:
    files=shared.Files(
        content=f.read(),
        file_name=filename,
    )

req = shared.PartitionParameters(files=files)
```

Once the API call is made, transform the result into a readable JSON format:

```
try:
    resp = s.general.partition(req)
except SDKError as e:
    print(e)

JSON(json.dumps(resp.elements[0:3], indent=2))
```

4.3.3 Find Elements Associated with Chapters

In this step, we want to find all chapters by filtering the "type" key to be "Title". Additionally, we add a filter to search only for titles related to hockey by applying another filter on the title text:

```
[x for x in resp.elements if x['type'] == 'Title' and 'hockey' in x['text'].lower()]
```

Now suppose we've extracted chapter names, for example, from the table of contents:

```
chapters = [
    "THE SUN-SEEKER",
    "RINKS AND SKATERS",
    "TEES AND CRAMPITS",
    "ICE-HOCKEY",
    "SKI-ING",
    "NOTES ON WINTER RESORTS",
    "FOR PARENTS AND GUARDIANS",
]
```

With the chapter names, loop through all extracted elements to find the element IDs associated with those chapter titles:

```
chapter_ids = {}
for element in resp.elements:
    for chapter in chapters:
        if element["text"] == chapter and element["type"] == "Title":
            chapter_ids[element["element_id"]] = chapter
            break
```

The element ID for those titles will show up as a parent ID for elements within that chapter. This enables us to identify elements within a chapter, allowing hybrid search on a chapter. The previous code created a dictionary with element IDs as keys and chapter titles as values.

Once we have that mapping, we can easily convert parent IDs to chapter names:

```
chapter_to_id = {v: k for k, v in chapter_ids.items()}
[x for x in resp.elements if x["metadata"].get("parent_id") == chapter_to_id["ICE-HOCKEY"]][0]
```

4.3.4 Load Documents into a Vector Database

After identifying the document elements and determining the chapter each element belongs to, the next step is to load all this content into our vector database to perform hybrid searches.

In this case, we'll use *Chroma* as our vector database because it runs in memory and is a generally nice vector database.

First, set up our *Chroma* database:

```
client = chromadb.PersistentClient(path="chroma_tmp", settings=chromadb.Settings(allow_reset=True))
client.reset()
```

Then, set up a collection:

```
collection = client.create_collection(
    name="winter_sports",
    metadata={"hnsw:space": "cosine"}
)
```

We pass a couple of pieces of information to the `create_collection` function: the name and some metadata. This isn't the document-extracted metadata but metadata for the vector database itself, specifying to use cosine similarity within this particular vector space for the similarity search.

Once set up, use our chapter mapping to insert documents into the vector database with chapter metadata:

```python
for element in resp.elements:
    parent_id = element["metadata"].get("parent_id")
    chapter = chapter_ids.get(parent_id, "")
    collection.add(
        documents=[element["text"]],
        ids=[element["element_id"]],
        metadatas=[{"chapter": chapter}]
    )
```

Elements within a particular chapter have a parent ID corresponding to the chapter title's element ID. The mapping function provides the chapter name to search on, and loading it into the vector database uses the chapter name as metadata.

Now that all elements are loaded into the vector database, we can view them:

```python
results = collection.peek()
print(results["documents"])
```

4.3.5 Perform a Hybrid Search with Metadata

Now that the elements are loaded into the vector database with metadata, we can perform a hybrid search with metadata.

First, set up a query with query texts. Perform a hybrid search by conditioning that search on content occurring only in the ice hockey chapter:

```python
result = collection.query(
    query_texts=["How many players are on a team?"],
    n_results=2,
    where={"chapter": "ICE-HOCKEY"},
)
print(json.dumps(result, indent=2))
```

4.3.6 Chunking Content

We start by deserializing the serialized JSON content with document element information. Use the `dict_to_elements` function from the unstructured library:

```python
elements = dict_to_elements(resp.elements)
```

Once deserialized, chunk this content using the `chunk_by_title` function, operating on the elements:

```python
chunks = chunk_by_title(
    elements,
    combine_text_under_n_chars=100,
    max_characters=3000,
)
```

The `combine_text_under_n_chars` setting combines chunks if an element has fewer than 100 characters, preventing very small chunks. The `max_characters` setting places a character limit on chunk size; exceeding this limit splits into a new chunk.

After chunking, view the result:

```python
JSON(json.dumps(chunks[0].to_dict(), indent=2))
```

5 Preprocessing PDFs and Images

Some document types like PDFs and images require models to preprocess. In this section, we'll learn about document image analysis techniques such as document layout detection and vision transformers, and how to use these techniques to process PDFs and images.

5.1 Document Image Analysis

So far, we've learned how to preprocess documents that require rule-based parsers. However, some documents don't have structured information available within the document itself, such as PDFs and images. For these documents, we need to use visual information to understand the structure of the document.

Document image analysis (DIA) allows us to *extract formatting information* and text *from the raw image* of a document. We'll learn about two DIA methods: Document Layout Detection and Vision Transformers.

5.1.1 Document Layout Detection

Document Layout Detection (DLD) uses an object detection model to draw and label bounding boxes around layout elements on a document image. Once it's drawn those bounding boxes, the boxes get labeled and then the text gets extracted from within the bounding box.

DLD requires two steps:

1. *Identifying and categorizing bounding boxes* around elements within the document using a computer vision model such as YOLOX or Detectron2. This involves drawing bounding boxes around each element within the document, such as narrative text, titles, or bulleted lists.
2. *Extracting text from within those bounding boxes*. Depending on the document type, there are two ways of doing this. In some cases, the text is not available from within the document itself, and we'll need to apply techniques such as Optical Character Recognition (OCR) to extract the text. In other cases, such as in some PDFs, the text is available within the document itself and can be extracted directly without OCR. We can use bounding box information to trace the bounding box back to the original document and extract the text content that falls within the bounding box.

5.1.2 Vision Transformers

Vision Transformer (ViT) models take a document image as an input and produce a text representation of a structured output (like JSON) as output. Vision transformers can optionally include a text prompt, similar to an LLM transformer.

In contrast to DLD, ViT extracts content from PDFs and images in a single step. Whereas DLD models will draw a bounding box and then apply, when necessary, an OCR model, ViT *accepts images as input and then produces text directly as output*. In this case, OCR is not required to extract the text from the image.

One common architecture for ViT is the *Donut architecture* (Document Understanding Transformer). When applying a model such as the Donut model, we can train the model to produce a valid JSON string as output, and that JSON string can contain the structured document output that we're interested in.

5.1.3 When to Use DLD or ViT Models

When should we use a DLD model and when a ViT model? Each model type has its advantages and disadvantages.

For DLD models:

- *Advantages:* The model is trained on a fixed set of element types and can become very good at recognizing those. Additionally, we can get bounding box information, which allows us to trace the results back to the original document and, in some cases, extract text without running OCR.
- *Disadvantages:* DLD models often require two model calls (the object detection and the OCR models). These models are also less flexible as they work from a fixed set of element types, so if we need to extract something else, they may not be able to do that.

For ViT models:

- *Advantages:* They are flexible for non-standard document types like forms and can easily extract information like key-value pairs. They are also more adaptable to new ontologies, where adding a new element type is simpler, potentially through prompting.
- *Disadvantages:* The model is generative and can be prone to hallucination or repetition, similar to generative models in natural language use cases. These models are also much more computationally expensive than DLD models, requiring significant computing power or running more slowly.

5.2 Practical Exercise

Let's apply these techniques in practice. In this exercise, we'll preprocess the same document first in an HTML representation and then in a PDF representation. We'll see how to extract a similar set of document types whether we're preprocessing the document using rule-based techniques or extracting that information based on visual cues.

5.2.1 General Code

First, we'll need to import some dependencies as we've done before:

```python
# Warning control
import warnings
warnings.filterwarnings('ignore')

from unstructured_client import UnstructuredClient
from unstructured_client.models import shared
from unstructured_client.models.errors import SDKError

from unstructured.partition.html import partition_html
from unstructured.partition.pdf import partition_pdf

from unstructured.staging.base import dict_to_elements
```

Next, set up the connection to the unstructured API:

```python
from Utils import Utils
utils = Utils()

DLAI_API_KEY = utils.get_dlai_api_key()
DLAI_API_URL = utils.get_dlai_url()

s = UnstructuredClient(
    api_key_auth=DLAI_API_KEY,
    server_url=DLAI_API_URL,
)
```

5.2.2 Preprocess as HTML

First, process the document as HTML. We don't need to call the API for this:

```python
filename = "example_files/el_nino.html"
html_elements = partition_html(filename=filename)

for element in html_elements[:10]:
    print(f"{element.category.upper()}: {element.text}")
```

5.2.3 Preprocess with Fast Strategy

Now, preprocess the PDF file using the fast strategy from the unstructured library. The fast strategy extracts text directly from the document and can be used for simple PDFs:

```python
filename = "example_files/el_nino.pdf"
pdf_elements = partition_pdf(filename=filename, strategy="fast")

for element in pdf_elements[:10]:
    print(f"{element.category.upper()}: {element.text}")
```

Running this code, we'll see that the elements found are very similar to those from the HTML processing.

5.2.4 Preprocess with DLD Model

For more complex PDFs, use the API to preprocess the document using the YOLOX model, which will draw bounding boxes around each element and then extract the text within them. Supply this information to the unstructured API using the `hi_res_model_name` parameter:

```python
with open(filename, "rb") as f:
    files=shared.Files(
        content=f.read(),
        file_name=filename,
    )
```

```python

req = shared.PartitionParameters(
    files=files,
    strategy="hi_res",
    hi_res_model_name="yolox",
)

try:
    resp = s.general.partition(req)
    dld_elements = dict_to_elements(resp.elements)
except SDKError as e:
    print(e)
```

Once the API call has completed, we can see the output, which is very similar to the HTML and fast strategy output:

```python
for element in dld_elements[:10]:
    print(f"{element.category.upper()}: {element.text}")
```

5.2.5 Compare Outputs

Once we have these outputs, compare them to see how similar they are:

```python
import collections

len(html_elements)
html_categories = [el.category for el in html_elements]
collections.Counter(html_categories).most_common()

len(dld_elements)
dld_categories = [el.category for el in dld_elements]
collections.Counter(dld_categories).most_common()
```

So, whether this document is represented in PDF or HTML form, we'll get almost the same output and can treat the documents the same in our application.

6 Extracting Tables

Tables are important elements of many documents. In this section, we'll learn how to extract tables from documents and infer their structure.

6.1 Table Extraction

Most RAG use cases focus on text content within documents. However, in some industries, it is common to see structured data embedded in unstructured documents. This often occurs in industries such as finance and insurance. Table extraction enables RAG applications to extract information contained within tables in unstructured documents.

Some document types, such as HTML and Word, contain table structure information within the document itself. For these document types, we can use rules-based parsers to extract table information. For other document types such as PDFs and images, we need to use visual cues to first identify the table within the document and then process it to extract information from the table.

There are a few techniques we'll learn about to accomplish this: table transformers, vision transformers, and OCR preprocessing. Once we've extracted the content from the table, we convert the table to HTML output so we can preserve the structure when we pass the table to an LLM.

6.1.1 Table Transformers

A table transformer is a model that identifies bounding boxes for table cells and then converts the output to HTML. There are two steps in this process:

1. First, identify tables using the DLD model.
2. Then, run the table through the table transformer.

There are some advantages to this technique, including the ability to trace cells back to the original bounding boxes. We have bounding box information about each individual cell within the table.

The disadvantages include multiple expensive model calls, including multiple DLD calls and then multiple OCR calls.

6.1.2 Vision Transformers

We can also extract table content from PDFs and images using vision transformers like the ones seen in the previous section. However, unlike the previous section when the target output was JSON, in this case, the target output will be HTML.

An advantage of this method is that it allows for prompting, is more flexible, and only involves a single model call. The disadvantage is that it's generative, prone to hallucination, and we don't retain any bounding box information.

6.1.3 OCR Postprocessing

In this technique, we identify the tables using a DLD model. However, in this case, we'll OCR the table and then process the table using rules-based methods.

An advantage is that it's fast, and for well-behaved tables, it can be very accurate. Disadvantages include that it requires statistical or rules-based parsing, making it less flexible than other approaches. For more complex tables, the results may not be as good.

6.2 Practical Exercise

Now let's apply these techniques to extract information from some tables contained in a PDF.

6.2.1 General Code

As usual, import the necessary helper functions and set up the connection to the unstructured API:

```python
# Warning control
import warnings
warnings.filterwarnings('ignore')

from unstructured_client import UnstructuredClient
from unstructured_client.models import shared
from unstructured_client.models.errors import SDKError

from unstructured.staging.base import dict_to_elements

from Utils import Utils
utils = Utils()
```

```

DLAI_API_KEY = utils.get_dlai_api_key()
DLAI_API_URL = utils.get_dlai_url()

s = UnstructuredClient(
    api_key_auth=DLAI_API_KEY,
    server_url=DLAI_API_URL,
)
```

6.2.2 Extracting Tables

Now, preprocess a PDF that includes tables and images.

To preprocess the document, pass it to the unstructured API. Note a few parameters: the `pdf_infer_table_structure` parameter tells the API that we want to extract table information, and the `skip_infer_table_types` parameter instructs the API not to skip table extraction for any document types:

```python
filename = "example_files/embedded-images-tables.pdf"

with open(filename, "rb") as f:
    files=shared.Files(
        content=f.read(),
        file_name=filename,
    )

req = shared.PartitionParameters(
    files=files,
    strategy="hi_res",
    hi_res_model_name="yolox",
    skip_infer_table_types=[],
    pdf_infer_table_structure=True,
)

try:
    resp = s.general.partition(req)
    elements = dict_to_elements(resp.elements)
except SDKError as e:
    print(e)
```

Once the API has completed the request, filter down to just the tables in the document using a filter operation similar to what we've seen earlier. In this case, look for elements with the category "Table":

```python
tables = [el for el in elements if el.category == "Table"]
```

Now that we have the table elements, we can see the text of the table using the `text` attribute on the element:

```python
tables[0].text
```

As mentioned earlier, it's also helpful to have an HTML representation of the table so that we can pass the information to an LLM while maintaining the table structure. This information is available in the `text_as_html` field within the element metadata for tables:

```python
table_html = tables[0].metadata.text_as_html
```

We can use the following code block to view what the HTML in the metadata field looks like:

```python
from io import StringIO
from lxml import etree

parser = etree.XMLParser(remove_blank_text=True)
file_obj = StringIO(table_html)
tree = etree.parse(file_obj, parser)
print(etree.tostring(tree, pretty_print=True).decode())
```

We can also display the table:

```python
from IPython.core.display import HTML
HTML(table_html)
```

Once we've extracted the table content from the document and converted the table content to HTML, it can be helpful to summarize these tables so that we can search on these tables when we perform a similarity search within a RAG architecture.

To do this, we'll use a few utilities from *LangChain*, including a helper function for summarizing:

```python
from langchain_openai import ChatOpenAI
from langchain_core.documents import Document
from langchain.chains.summarize import load_summarize_chain

llm = ChatOpenAI(temperature=0, model_name="gpt-3.5-turbo-1106")
chain = load_summarize_chain(llm, chain_type="stuff")
chain.invoke([Document(page_content=table_html)])
```

Once we've imported the helper functions, we can instantiate the summarization chain and summarize the table HTML content.

7 Build Your Own RAG Bot

Now that we've learned all these techniques, in this section we'll put them all together and build a RAG bot over a corpus including PDF, PowerPoint, and Markdown documents.

We'll preprocess all these document types, load them in a vector database, query the database, insert the result of a query into a prompt, and then pass the prompt to an LLM.

7.1 General Code

We can start by, again, importing a few helper functions and setting up the connection to the API:

```python
# Warning control
import warnings
warnings.filterwarnings('ignore')

from unstructured_client import UnstructuredClient
from unstructured_client.models import shared
from unstructured_client.models.errors import SDKError

from unstructured.chunking.title import chunk_by_title
from unstructured.partition.md import partition_md
from unstructured.partition.pptx import partition_pptx
from unstructured.staging.base import dict_to_elements

import chromadb

from Utils import Utils
utils = Utils()

DLAI_API_KEY = utils.get_dlai_api_key()
DLAI_API_URL = utils.get_dlai_url()

s = UnstructuredClient(
    api_key_auth=DLAI_API_KEY,
    server_url=DLAI_API_URL,
)
```

We have added a new function useful for handling Markdown files.

7.2 Preprocess the PDF

We use the unstructured API to call the YOLOX model to preprocess the document using a DLD model:

```python
filename = "example_files/donut_paper.pdf"

with open(filename, "rb") as f:
    files=shared.Files(
        content=f.read(),
        file_name=filename,
    )

req = shared.PartitionParameters(
    files=files,
    strategy="hi_res",
    hi_res_model_name="yolox",
    pdf_infer_table_structure=True,
    skip_infer_table_types=[],
)

try:
    resp = s.general.partition(req)
    pdf_elements = dict_to_elements(resp.elements)
except SDKError as e:
    print(e)
```

Now if we want, we can look at the first element obtained, in this case, the header:

```python
pdf_elements[0].to_dict()
```

We may also want to take a look at some of the tables we'll be able to query over:

```
tables = [el for el in pdf_elements if el.category == "Table"]
table_html = tables[0].metadata.text_as_html
```

This PDF also contains some information that we may not be interested in querying over. We can use some of the metadata we've extracted while preprocessing to filter out unwanted content.

7.2.1 Filter References Section

First, we can filter out the references section. To find elements that belong to the references section, we can find elements that are nested under the references element by using the parent ID metadata field.

First, get the reference title:

```
reference_title = [
    el for el in pdf_elements
    if el.text == "References"
    and el.category == "Title"
][0]
```

When we convert the reference title element to a dictionary, we can see the element ID; that's what we can use to filter out elements that belong to the references section:

```
reference_title.to_dict()
```

We can save that ID, then look for elements that have that element ID as the parent ID:

```
references_id = reference_title.id

pdf_elements = [el for el in pdf_elements if el.metadata.parent_id != references_id]
```

Now our element set does not contain any elements from the references section.

7.2.2 Filter out Headers

Another part that we might want to remove is the headers, the text at the top of the page that often contains the section title.

To remove them, we can simply filter on the metadata field category to remove the headers from the output:

```
headers = [el for el in pdf_elements if el.category == "Header"]

pdf_elements = [el for el in pdf_elements if el.category != "Header"]
```

7.3 Preprocess the PowerPoint

To preprocess this file, simply call the `partition_pptx` function:

```
filename = "example_files/donut_slide.pptx"
pptx_elements = partition_pptx(filename=filename)
```

7.4 Preprocess the README

Similarly, we can partition the Markdown file by calling the `partition_md` function:

```
filename = "example_files/donut_readme.md"
md_elements = partition_md(filename=filename)
```

7.5 Load the Documents into the Vector DB

Now that we've preprocessed all of our documents, we can combine them into a single corpus and chunk them using the `chunk_by_title` function:

```
elements = chunk_by_title(pdf_elements + pptx_elements + md_elements)
```

After chunking, we can load the documents into a vector database. In this case, we can use utilities from *LangChain* to handle the loading into the database:

```
from langchain_community.vectorstores import Chroma
from langchain_core.documents import Document
from langchain_openai import OpenAIEmbeddings
```

For this app, we may want to search for content that belongs to a specific file type within the corpus. So, when we load the documents into the vector DB, we can include the source as a metadata field:

```
documents = []
for element in elements:
    metadata = element.metadata.to_dict()
    del metadata["languages"]
    metadata["source"] = metadata["filename"]
    documents.append(Document(page_content=element.text, metadata=metadata))
```

Once we've created documents to load into the vector database, the next step is to embed those documents, in this case using OpenAI embeddings. Then we can use the `from_documents` method on the *Chroma* database object to load the documents:

```
embeddings = OpenAIEmbeddings()

vectorstore = Chroma.from_documents(documents, embeddings)
```

When the documents load, they'll run through the embedding process and then once they're embedded, they'll be uploaded into the *Chroma* vector database.

Now we can set up a retriever to search over the database:

```
retriever = vectorstore.as_retriever(
    search_type="similarity",
    search_kwargs={"k": 6}
)
```

In this case, we'll search on similarity and retrieve six results before building our prompt to pass to the LLM.

Now the vector database is set up. The next step is to set up a prompt template. In this case, we can use *LangChain* to help manage our prompt template:

```
from langchain.prompts.prompt import PromptTemplate
from langchain_openai import OpenAI
from langchain.chains import ConversationalRetrievalChain, LLMChain
from langchain.chains.qa_with_sources import load_qa_with_sources_chain

template = """You are an AI assistant for answering questions about the Donut document understanding model.
You are given the following extracted parts of a long document and a question. Provide a conversational answer.
If you don't know the answer, just say "Hmm, I'm not sure." Don't try to make up an answer.
If the question is not about Donut, politely inform them that you are tuned to only answer questions about Donut.
Question: {question}
=========
{context}
=========
Answer in Markdown:"""
prompt = PromptTemplate(template=template, input_variables=["question", "context"])
```

In this case, we've created a prompt that instructs the LLM to say "I don't know" if it doesn't know the answer to the question we're asking.

Now we're ready to query the LLM. Simply load the `ConversationalRetrievalChain` from *LangChain*:

```
llm = OpenAI(temperature=0)

doc_chain = load_qa_with_sources_chain(llm, chain_type="map_reduce")
question_generator_chain = LLMChain(llm=llm, prompt=prompt)
qa_chain = ConversationalRetrievalChain(
    retriever=retriever,
    question_generator=question_generator_chain,
    combine_docs_chain=doc_chain,
)
```

Now that we've instantiated the chain, we can ask a question:

```
qa_chain.invoke({
    "question": "How does Donut compare to other document understanding models?",
    "chat_history": []
})["answer"]
```

In addition to the answer to the question, because we included metadata about the file name when we uploaded the documents to the vector database, the model can cite the source of this information.

We may also be interested in finding information from a specific source within the corpus. To accomplish this, we can apply a filter on the source:

```
filter_retriever = vectorstore.as_retriever(
    search_type="similarity",
    search_kwargs={"k": 1, "filter": {"source": "donut_readme.md"}}
)
```

With that retriever, we can set up a new chain and then run a new query:

```
filter_chain = ConversationalRetrievalChain(
    retriever=filter_retriever,
    question_generator=question_generator_chain,
    combine_docs_chain=doc_chain,
)

filter_chain.invoke({
    "question": "How do I classify documents with DONUT?",
    "chat_history": [],
    "filter": filter,
})["answer"]
```

Now the model has responded with content that was contained within the README for the Donut model.