How to Feed Your RAG: Ingesting and Structuring Your Data

We're diving into what many experts consider the most critical yet most overlooked aspect of RAG systems: data ingestion and knowledge base preparation.

Why Your RAG System Is Only as Good as Your Data Pipeline

"Garbage in, garbage out" applies more to RAG than almost any other AI system. You could have the most sophisticated retrieval algorithms and the most powerful LLM, but with poorly prepared data, your outputs will disappoint every time.

The Complete Data Ingestion Pipeline

Let me walk you through the complete data pipeline with practical examples:

1. Data Sources: What Can You Feed Into RAG?

RAG systems can ingest virtually any text-based information:

- **Documents:** PDFs, Word docs, PowerPoints, Excel files
- **Web content:** HTML pages, wikis, knowledge bases
- **Databases:** SQL, NoSQL records, CSV files
- **Communication:** Emails, chat logs, support tickets
- **Code:** Repositories, documentation, comments
- **Media transcripts:** Meeting recordings, videos, podcasts

Text Loaders – Extract Text from 5 Different Sources

In many real-world applications, you'll need to extract text data from various file formats. Below are five different types of document loaders implemented using Python — without using LangChain. Each section includes a clear explanation, necessary installation commands, and clean Python code.

1. PDF Text Loader

Explanation:

- PDF files are widely used for documents such as reports, research papers, and books.
- We use the PyPDF2 library to load and read these files.
- The code reads all pages in the PDF and extracts the text using `.extract_text()`.

Required Installation:

```
pip install PyPDF2
```

Code:

```
from PyPDF2 import PdfReader

def load_pdf(file_path):
    pdf_reader = PdfReader(file_path)
    text = ""
    for page in pdf_reader.pages:
        text += page.extract_text()
    return text

file_path = "documents/sample_document.pdf"
pdf_text = load_pdf(file_path)
print(pdf_text)
```

2. DOCX Text Loader (Microsoft Word)

Explanation:

- DOCX files are commonly used for resumes, letters, and reports.
- We use the python-docx library to read the file.
- The code iterates through all paragraphs and concatenates them into a single string.

Required Installation:

```
pip install python-docx
```

Code:

```
from docx import Document

def load_docx(file_path):
    doc = Document(file_path)
    text = ""
    for paragraph in doc.paragraphs:
        text += paragraph.text + "\n"
    return text

file_path = "documents/sample_resume.docx"
docx_text = load_docx(file_path)
print(docx_text)
```

3. Web Page Text Loader

Explanation:

- Web pages often contain useful information in HTML format.
- We use the requests library to fetch the HTML content and BeautifulSoup to parse and extract clean text.
- The code removes HTML tags and returns only the visible text.

Required Installation:

```
pip install requests
pip install beautifulsoup4
```

Code:

```
import requests
from bs4 import BeautifulSoup

def load_web_text(url):
    response = requests.get(url)
    soup = BeautifulSoup(response.text, 'html.parser')
    text = soup.get_text()
    return text

url = "https://example.com"
web_text = load_web_text(url)
print(web_text)
```

4. Excel and CSV Text Loader

Explanation:

- Excel (.xlsx) and CSV (.csv) files are widely used for structured data.
- We use the pandas library to read both formats.

- The code checks the file extension and uses the appropriate reader (`read_excel` or `read_csv`), then converts the DataFrame to a string.

Required Installation:

```
pip install pandas
```

Code:

```
import pandas as pd

def load_excel_csv(file_path):
    if file_path.endswith('.xlsx'):
        df = pd.read_excel(file_path)
    else:
        df = pd.read_csv(file_path)
    text = df.to_string(index=False)
    return text

file_path = "data/sample_dataset.xlsx"
excel_csv_text = load_excel_csv(file_path)
print(excel_csv_text)
```

5. JSON Text Loader

Explanation:

- JSON is a lightweight format for storing and exchanging structured data.
- Python's built-in `json` module can read and parse JSON files.
- The code loads the JSON file and converts it to a readable string using `json.dumps`.

Required Installation:

No additional installation is required. Python includes the `json` module by default.

Code:

```
import json

def load_json(file_path):
    with open(file_path, 'r') as file:
        data = json.load(file)
    text = json.dumps(data, indent=4)
    return text

file_path = "data/sample_config.json"
json_text = load_json(file_path)
print(json_text)
```

2. Cleaning & Preprocessing: The Unsung Hero

After loading text from various sources like PDF, DOCX, websites, Excel/CSV, and JSON files, the next critical step is text cleaning and preprocessing. This ensures the data is usable for downstream tasks such as search, training, or analysis.

Below is a breakdown of what to clean, why it's important, and practical code examples.

Why Cleaning is Important

Unclean text can cause:

- Noise in vectorization or model training (e.g., unnecessary symbols).
- Inconsistent results due to formatting issues (e.g., line breaks, bullets).
- Lower performance in tasks like classification, similarity search, and summarization.

Common Cleaning Tasks

1. Remove Extra Whitespaces, Tabs, and Newlines

Unwanted whitespace can interrupt sentence flow.

```
def remove_whitespace(text):
    return " ".join(text.split())

# Example usage (assuming raw_text is defined from a loader)
# raw_text = "   This is   a sample text with\n   extra   spaces and\t tabs.   "
# cleaned_text = remove_whitespace(raw_text)
# print(cleaned_text)
```

2. Remove Special Characters and Symbols

Special characters may not carry semantic value and can clutter the text.

```
import re

def remove_special_characters(text):
    return re.sub(r'[^a-zA-Z0-9\s]', '', text)

# Example usage (assuming cleaned_text is defined)
# cleaned_text = remove_special_characters(cleaned_text)
# print(cleaned_text)
```

3. Lowercase the Text

Helps normalize the text for matching, tokenizing, and embedding.

```
def to_lowercase(text):
    return text.lower()

# Example usage (assuming cleaned_text is defined)
# cleaned_text = to_lowercase(cleaned_text)
# print(cleaned_text)
```

4. Remove Stopwords (Optional but useful in NLP)

Stopwords like "and", "the", "is" can be removed to reduce noise.

```
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
import nltk

# Download necessary NLTK data (run once)
try:
    nltk.data.find('tokenizers/punkt')
except nltk.downloader.DownloadError:
    nltk.download('punkt')
try:
    nltk.data.find('corpora/stopwords')
except nltk.downloader.DownloadError:
    nltk.download('stopwords')

def remove_stopwords(text):
    tokens = word_tokenize(text)
    stop_words = set(stopwords.words('english'))
    filtered_tokens = [word for word in tokens if word.lower() not in stop_words]
    return " ".join(filtered_tokens)

# Example usage (assuming cleaned_text is defined)
# cleaned_text = remove_stopwords(cleaned_text)
# print(cleaned_text)
```

5. Combine All Steps into a Single Pipeline

```
import re
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
import nltk

# Ensure NLTK data is downloaded
try:
    nltk.data.find('tokenizers/punkt')
except nltk.downloader.DownloadError:
    nltk.download('punkt')
try:
    nltk.data.find('corpora/stopwords')
except nltk.downloader.DownloadError:
    nltk.download('stopwords')
```

```
def remove_whitespace(text):
    return " ".join(text.split())

def remove_special_characters(text):
    return re.sub(r'[^a-zA-Z0-9\s]', '', text)

def to_lowercase(text):
    return text.lower()

def remove_stopwords(text):
    tokens = word_tokenize(text)
    stop_words = set(stopwords.words('english'))
    filtered_tokens = [word for word in tokens if word.lower() not in stop_words]
    return " ".join(filtered_tokens)

def clean_text_pipeline(text):
    text = remove_whitespace(text)
    text = remove_special_characters(text)
    text = to_lowercase(text)
    text = remove_stopwords(text)
    return text

# Example usage:
# raw_text = "   This is   a sample text with\n   extra   spaces and\t tabs. It also has some !@#$%^
# final_cleaned_text = clean_text_pipeline(raw_text)
# print(final_cleaned_text)
```

Things to Keep in Mind While Cleaning

Aspect	Consideration
Preserve Important Terms	Avoid removing domain-specific keywords or acronyms
Unicode Handling	Normalize text to avoid issues with accented characters
Language Awareness	Stopwords and punctuation vary by language — ensure language detection first
Data Purpose	Don't over-clean if raw formatting is needed for context or layout analysis

3️⃣ Chunking Strategies – Breaking Large Text into Meaningful Parts

Once the text is cleaned and preprocessed, the next important step is chunking. Large documents can't be processed all at once by many models (especially LLMs) due to token limits. So we divide them into smaller, manageable "chunks" of text.

Why Chunking is Necessary

- Overcome model token limits – LLMs like GPT have token limits (e.g., 4096 or 8192 tokens).
- Improve semantic understanding – Chunking preserves context better than sentence-level processing.
- Enable retrieval-augmented generation (RAG) – For vector search or QA systems.

What to Consider Before Chunking

Factor	Description
Chunk size	How large should each chunk be (in characters or tokens)?
Overlap	Should chunks overlap to preserve context between them?
Separator	Chunk based on sentences, paragraphs, or fixed-length characters?
Format awareness	Maintain structure where needed (like section titles or bullet points)

1. Fixed-Size Character-Based Chunking

Divides text into fixed character lengths with optional overlap.

```
def character_chunking(text, chunk_size=1000, overlap=200):
    chunks = []
    start = 0
    while start < len(text):
        end = start + chunk_size
        chunks.append(text[start:end])
        start += chunk_size - overlap
    return chunks

# Example usage (assuming final_cleaned_text is defined from cleaning pipeline)
# final_cleaned_text = "This is a very long text that needs to be chunked. It contains multiple sent
# chunks = character_chunking(final_cleaned_text)
# print(f"Total Chunks: {len(chunks)}")
# print(chunks[0])
```

2. Sentence-Based Chunking

Divides text by sentences and groups them until the chunk size is met.

```
import nltk
from nltk.tokenize import sent_tokenize

# Download necessary NLTK data (run once)
try:
    nltk.data.find('tokenizers/punkt')
except nltk.downloader.DownloadError:
    nltk.download('punkt')

def sentence_chunking(text, max_chars=1000):
    sentences = sent_tokenize(text)
    chunks = []
    current_chunk = ""

    for sentence in sentences:
        # Check if adding the next sentence exceeds max_chars
        # Add 1 for the space that will separate sentences
        if len(current_chunk) + len(sentence) + (1 if current_chunk else 0) <= max_chars:
            if current_chunk:
                current_chunk += " " + sentence
            else:
                current_chunk = sentence
        else:
            chunks.append(current_chunk.strip())
            current_chunk = sentence
    if current_chunk: # Add the last chunk if it's not empty
        chunks.append(current_chunk.strip())
    return chunks

# Example usage (assuming final_cleaned_text is defined)
# final_cleaned_text = "This is the first sentence. This is the second sentence. This is a much long
# chunks = sentence_chunking(final_cleaned_text)
# print(f"Total Chunks: {len(chunks)}")
# print(chunks[0])
```

3. Token-Based Chunking (Using tiktoken)

Token-level chunking gives more control when working with LLMs like GPT.

```
import tiktoken

def token_chunking(text, max_tokens=300, overlap=50, model_name="gpt-3.5-turbo"):
    encoding = tiktoken.encoding_for_model(model_name)
    tokens = encoding.encode(text)

    chunks = []
    start = 0
    while start < len(tokens):
        end = start + max_tokens
        chunk_tokens = tokens[start:end]
        chunk_text = encoding.decode(chunk_tokens)
        chunks.append(chunk_text)
        start += max_tokens - overlap
    return chunks

# Example usage (assuming final_cleaned_text is defined)
# final_cleaned_text = "This is a sample text for token-based chunking. It will be encoded into toke
# chunks = token_chunking(final_cleaned_text)
```

```
# print(f"Total Chunks: {len(chunks)}")
# print(chunks[0])
```

4. Paragraph-Based Chunking

Ideal when paragraphs carry logical structure (e.g., articles, reports).

```
def paragraph_chunking(text, max_chars=1000):
    paragraphs = text.split('\n\n') # Split by double newline for paragraphs
    chunks = []
    current_chunk = ""

    for para in paragraphs:
        # Add 2 for the double newline separator if current_chunk is not empty
        if len(current_chunk) + len(para) + (2 if current_chunk else 0) <= max_chars:
            if current_chunk:
                current_chunk += "\n\n" + para
            else:
                current_chunk = para
        else:
            chunks.append(current_chunk.strip())
            current_chunk = para
    if current_chunk: # Add the last chunk if it's not empty
        chunks.append(current_chunk.strip())
    return chunks

# Example usage (assuming final_cleaned_text is defined)
# final_cleaned_text = """
# This is the first paragraph. It talks about a general topic.
#
# This is the second paragraph. It delves into more specific details related to the first topic.
#
# A third paragraph follows, introducing a new sub-topic. This paragraph might be quite long, potent
#
# Finally, a fourth paragraph concludes the section.
# """ * 5
# chunks = paragraph_chunking(final_cleaned_text)
# print(f"Total Chunks: {len(chunks)}")
# print(chunks[0])
```

5. Line-Based Chunking

Sometimes used for line-by-line analysis or transcript processing.

```
def line_chunking(text, max_lines=10):
    lines = text.splitlines()
    chunks = []
    for i in range(0, len(lines), max_lines):
        chunk = "\n".join(lines[i:i+max_lines])
        chunks.append(chunk)
    return chunks

# Example usage (assuming final_cleaned_text is defined)
# final_cleaned_text = """
# Line 1: This is a sample line.
# Line 2: Another line of text.
```

```
# Line 3: And another one.
# Line 4: This could be a log entry.
# Line 5: Or a line from a script.
# Line 6: Each line is distinct.
# Line 7: For certain data types.
# Line 8: Line-based chunking is useful.
# Line 9: It maintains line integrity.
# Line 10: And is very straightforward.
# Line 11: This is the start of a new chunk.
# Line 12: And so on.
# """ * 3
# chunks = line_chunking(final_cleaned_text)
# print(f"Total Chunks: {len(chunks)}")
# print(chunks[0])
```

Summary Table – Chunking Types

Type	Best Use Case	Pros	Cons
Character	Generic documents, easy to implement	Fast and simple	May split in middle of sentence
Sentence	Summarization, QA	Preserves sentence context	Complex with very long sentences
Token	LLM-specific tasks	Aligns with token limits	Requires tokenizer
Paragraph	Reports, research papers	Preserves logical structure	Inconsistent paragraph length
Line	Code, logs, transcripts	Lightweight and fast	Not meaningful for narrative text

4 Understanding Metadata in Document Processing

Metadata refers to additional information associated with a document or its chunks that provides context beyond just the raw text. It plays a crucial role in improving the performance, relevance, and traceability of document-based applications.

What is Metadata?

Metadata is data about data. In text processing pipelines, it may include:

- File name or title
- Source (path or URL)
- Page number (for PDFs)
- Document type
- Creation or modification date
- Chunk index or character range
- Tags or categories

Why Metadata Matters

- **Improved Filtering and Retrieval**: Enables targeted search based on document type, date, or file name.
- **Better Context for LLMs**: Helps provide meaningful responses by identifying document structure or origin.
- **Source Traceability**: Ensures that outputs can be tracked back to specific parts of the source.
- **Organized Indexing**: Essential when storing vectors in vector databases like FAISS, Chroma, or Pinecone.

What Happens Without Metadata?

- Lack of filtering and relevance in search
- Hard to explain or trace LLM outputs
- Reduced model performance due to missing context
- No way to debug or audit responses

How to Add Metadata

Let's enhance the chunks created earlier (from Part 3) by adding metadata such as file name, chunk index, and character positions.

Code Snippet – Attaching Metadata to Chunks:

```
def add_metadata_to_chunks(chunks, file_name):
    chunks_with_metadata = []
    # Assuming character_chunking was used with chunk_size=1000, overlap=200
    # Adjust start_char calculation if a different chunking method is used
    for idx, chunk in enumerate(chunks):
        # This calculation assumes a fixed-size character chunking with overlap
        # For other chunking methods, you might need to track start/end chars differently
        start_char = idx * (1000 - 200) # Example: 1000 is chunk_size, 200 is overlap
        end_char = start_char + len(chunk)
        chunk_data = {
            "text": chunk,
            "metadata": {
                "file_name": file_name,
```

```
                "chunk_index": idx,
                "start_char": start_char,
                "end_char": end_char,
                "char_count": len(chunk)
            }
        }
        chunks_with_metadata.append(chunk_data)
    return chunks_with_metadata

# Example usage:
# Assuming 'chunks' is a list of text strings obtained from a chunking function
# For demonstration, let's create some dummy chunks:
# dummy_text = "This is a sample text. It needs to be chunked. Each chunk will have metadata. Metada
# dummy_chunks = ["This is a sample text. It needs to be chunked.", "Each chunk will have metadata.
# chunks_with_meta = add_metadata_to_chunks(dummy_chunks, "How_Vectors.pdf")

# # Preview first chunk with metadata
# if chunks_with_meta:
#     print("Text Preview:", chunks_with_meta[0]['text'][:150])
#     print("Metadata:", chunks_with_meta[0]['metadata'])
```

Why Can't We Just Convert Entire Documents to Embeddings?

Imagine you're building a library system. If you only cataloged entire books (not chapters or sections), then when someone asks for information about "photosynthesis":

- You'd have to retrieve entire biology textbooks (1000+ pages)
- The searcher would need to manually scan these books to find relevant sections
- Many books with brief mentions would score similarly to books with detailed chapters

Similarly, with RAG:

- **Relevance Dilution:** A 50-page document might have only one relevant paragraph. When embedded as a whole, that signal gets diluted.
- **Context Window Limitations:** LLMs have token limits (4K-32K). Whole documents often exceed these limits.
- **Retrieval Granularity:** You want to retrieve the most specific, relevant information - not entire documents.
- **Vector Similarity Issues:** Longer texts create "averaged out" embeddings that lose specificity.

Common Chunking Mistakes That Kill RAG Performance

Mistake #1: One-size-fits-all chunking

Different document types need different strategies. Legal contracts need semantic chunking by clauses, while news articles might work better with fixed-size chunks.

Mistake #2: Ignoring document structure

Headers, sections, and hierarchies contain valuable context. Don't blindly chunk across these boundaries.

Mistake #3: No chunk overlap

Without overlap,