

### Thursday, July 10 (7/10/25)

(Grant) Moved ChatGPT framework to PyCharm. Bug-fixed the framework by continually putting errors into ChatGPT. Created a folder structure manually to house all of the code. Switched the PDF reader from pdfplumber to PyMuPDF to index the larger, more dense lecture notes. 3000+ lecture notes were too big and “painfully malformed”, so switched to a 15 page example article instead. I switched encoding to fallback encoding (tries UTF-8, if not, tries Latin-1 encoding), for the LaTeX files. Switched the LLM to facebook/opt-1.3b for current setup, will switch to a better one later (Did not have access to mistralai/Mistral-7B-Instruct-v0.1). ChaGPT advised I used a local LLM due to the complexity of the files/notes being used (con: heavy resource requirements. Will have to use larger computers). Files successfully indexed. Downloaded even more packages, and added a debug print to make sure the code was running. Kept receiving “none” as an answer for every question, will debug tomorrow using ChatGPT.

### Friday, July 11 (7/11/25)

Grant: Continued bug-fixing to the point to which you could ask the model a question and it would answer based on a sample document provided (and DSL files). But, the current model could only answer questions that used up a small number of tokens (question and answer?). Attempted to use a different AI model, but each has different trade-offs. Needed to decide what LLM model to use either locally (small, but slower) or publicly (big, has lots of tokens, but need to pay for it). Due to needing to download models, get keys, and adapting the code for different models, I reverted it back to a small token amount question and answer bot (with facebook/opt-1.3b). In order to read the 3000+ document, we need to split it up into multiple separate documents and use Albert’s idea of running the retriever through ChatGPT. (Reminder: bot is a hybrid of structural and semantic indexing)

Rag file structure at end of day 2 of coding:

EngineeringRAGassistant/

```
├── app.py
├── rag_project.py
└── data/
    └── raw/
└── vector_index/
    └── index.pkl
└── ingest/
    ├── __init__.py
    ├── extract_pdf.py
    ├── extract_latex.py
    └── preprocess_pipeline.py
└── embed/
    └── embedder.py
```

```

├── retrieve/
│   └── retriever.py
└── generate/
    └── answerer.py

```

### Monday, July 14th (7/14/25)

Grant: Tried using ChatGPT from inside bot. Received API key from openAI, however I ran out of access to GPT 4o and tried using GPT 3.5 turbo instead. Ran into the problem of having to pay for more tokens in order to have the retriever and generator condensed into a single script. Decided to shift into only being a retriever to give me the chunks that were taken from the PDFs. I was able to turn the bot into only a receiver, then had output chunks turned into files that were transferred directly to my downloads.

The screenshot shows a file explorer window and a code editor side-by-side. The file explorer on the left displays the directory structure of the 'EngineeringRAGAssistant [rag\_project]' project. The code editor on the right shows the content of the 'answerer.py' file.

```

import os
from dotenv import load_dotenv
from openai import OpenAI

#load_dotenv()
#client = OpenAI(api_key=os.getenv("OPENAI_API_KEY"))

def answer_query(query, results):
    messages = [
        {"role": "system", "content": "You are a helpful assistant."},
        {"role": "user", "content": f"Use the following context to answer the question:\n\n{context}\n\nQuestion: {query}"}
    ]
    response = client.chat.completions.create(
        #model="gpt-3.5-turbo", # or "gpt-3.5-turbo" if needed
        #messages=messages,
        #temperature=0.7,
        #max_tokens=512,
    )
    #return response.choices[0].message.content.strip()
    return results

```

The screenshot shows a code editor window with the file 'rag\_project.py' open. The code defines a function 'query\_rag\_system' which performs a search using a retriever and then saves the retrieved context to a file.

```

from Utility.DocSaver import save_to_downloads
from embed.embedder import get_embedder
#from generate.answerer import answer_query # import the correct function
from retrieval.retriever import load_vectorstore
import os

embedder = get_embedder()
retriever = load_vectorstore()

def query_rag_system(question, save_sections=False, filename=None):
    query_vec = embedder.encode([question])[0]
    docs = retriever.search(query_vec, k=5)
    context = "\n\n".join([text for text, meta in docs])
    if not docs:
        return "No relevant context found."
    if save_sections and filename:
        # Build annotated text with metadata
        output_lines = []
        for i, (text, metadata) in enumerate(docs, start=1):
            output_lines.append(f"--- Section {i} ---")
            for key, value in metadata.items():
                output_lines.append(f"\t{key.capitalize()}: {value}")
            output_lines.append("")
            output_lines.append(text)
            output_lines.append("")
        full_content = "\n".join(output_lines)
        save_to_downloads(filename, "retrieved_context.txt", full_content)
    return docs

```

I then made the showing and downloading of the files optional. 1 for showing the chunks pulled from the question, or 2 for downloading the file directly to the downloads folder. I put the files directly into Copilot to see how the responses would turn out. It worked pretty well, and I will do more accuracy tests later. The current number of chunks as per the chunk setting is 5, but I can make it store more. Aditya recommended using DeepSeek AI because it may have a free API to potentially condense the bot into one script. I will do further research on that tomorrow. Tomorrow I plan to test the 3000+ page lecture notes and potentially try to generate DSL code using the DSL files by putting those files into ChatGPT/Copilot, etc.

Forgot to mention, I also added a utilities directory with DocSaver.py to the main file pipeline. I will specify the details of each file and how it works when the RAG bot is completed.

```
 1 # utils/saver.py
 2 import os
 3
 4 def save_to_downloads(filename: str, content: str): 2 usages
 5     downloads = os.path.join(os.path.expanduser("~/"), "Downloads")
 6     full_path = os.path.join(downloads, filename)
 7     with open(full_path, "w", encoding="utf-8") as f:
 8         f.write(content)
 9     print(f"[Saved] File saved to: {full_path}")
10
```

Tuesday, July 15th (7/15/25)

Grant:

Instructions as per Dr. Jeremic: Test DeepSeek, test inputting random/strange questions or characters, work on indexing, and validate output responses from AI

Personal Goals: Mess with topic indexer

DeepSeek API won't work, need to pay.

Tried remaking the index only for the 15 page example pdf to make testing easier, but chunks were still pulled from the other sources even when they were moved out of data/raw.

I made an index deleter so that any time you make a new index, it deletes the old one

The chunks pulled from the pdf are the same number as the number of pages, so I will try to make more, smaller chunks related to the sections of the paper.

Using the already imported fitz or PyMuPDF extractor tool, I made the chunks per section rather than per page. I moved the PDF extraction from extract\_pdf.py all to the preprocess\_pipeline and I still included the image, text, and equation metadata. I will test the functionality of this further.

```

1 # ingest/preprocess_pipeline.py
2 import os
3 import fitz # PyMuPDF
4 import pickle
5 from embed.embedder import embed_chunks
6 from embed.vector_store import VectorStore
7
8 VECTOR_DB_PATH = "./vector_index/index.pkl"
9
10 def extract_pdf_sections(filepath):  Usage
11     doc = fitz.open(filepath)
12     chunks = []
13
14     for page_num, page in enumerate(doc, start=1):
15         try:
16             blocks = page.get_text("dict")["blocks"]
17             images = [
18                 {"text": img[1], "xref": img[0]}
19                 for img in page.get_images(rutl=True)
20             ]
21
22             for i, block in enumerate(blocks):
23                 if "Times" not in block:
24                     continue
25
26                 text = ""
27                 fonts = set()
28                 max_font = 0
29
30                 for line in block["lines"]:
31                     for span in line["spans"]:
32                         if not span["text"].strip():
33                             continue
34                         text += span["text"] + " "
35                         fonts.add(span["font"].lower())
36                         max_font = max(max_font, span["size"])
37
38             is_bold = any("bold" in f for f in fonts)
39             is_heading = max_font >= 18 or is_bold
40
41             chunks.append({
42                 "text": text,
43                 "page": page_num,
44                 "source": os.path.basename(filepath),
45                 "font_size": max_font,
46                 "is_bold": is_bold,
47                 "type": "heading" if is_heading else "paragraph",
48                 "images": images if i == 0 else [],
49                 "equations": [] # placeholder if needed later
50             })
51
52         except Exception as e:
53             chunks.append({
54                 "text": "[Error extracting text]",
55                 "page": page_num,
56                 "source": os.path.basename(filepath),
57                 "type": "error",
58                 "error": str(e),
59                 "images": [],
60                 "equations": []
61             })
62
63     doc.close()
64
65     return chunks
66
67
68
69
70 def build_index():  2 usages
71     if os.path.exists(VECTOR_DB_PATH):
72         os.remove(VECTOR_DB_PATH)
73         print("Old vector index deleted.")
74
75     all_chunks, metadata = [], []
76
77     for file in os.listdir("data/raw"):
78         if not file.endswith(".pdf"):
79             continue
80
81         path = os.path.join("data/raw", file)
82         sections = extract_pdf_sections(path)
83
84         for i, section in enumerate(sections):
85             all_chunks.append(section["text"])
86             metadata.append({
87                 "chunk_id": f'{file}-p{section['page']}-{i+1}',
88                 "page": section["page"],
89                 "source": section["source"],
90                 "type": section["type"],
91                 "font_size": section["font_size"],
92                 "is_bold": section["is_bold"],
93                 "images": section["images"],
94                 "equations": section["equations"]
95             })
96
97         vectors = embed_chunks(all_chunks)
98         vs = VectorStore(dim=vectors.shape[1])
99         vs.add(vectors, all_chunks, metadata)
100
101         with open(VECTOR_DB_PATH, "wb") as f:
102             pickle.dump(vs, f)
103
104         print(f"✓ Index built with {len(all_chunks)} chunks.")

```

The console is still showing the chunks as per page rather than per section, even though the number of chunks increased from 15 to 143 (which may be too much). I will try to fix this.

I updated the code for semantic chunking per labeled section (as there are in scientific papers), however an attribute error occurred. I fixed the attribute error by skipping stray line breaks or malformed matches. I bugfixed some file errors, but still was retrieving irrelevant chunks from the PDF.

```

rag_project.py retriever.py topic_indexer.py DocSaver.py app.py preprocess_pipeline.py
1 # ingest/preprocess_pipeline.py
2 import os
3 import fitz # PyMuPDF
4 import pickle
5 import re
6 from embed.embedder import embed_chunks
7 from embed.vector_store import VectorStore
8
9 VECTOR_DB_PATH = "./vector_index/index.pkl"
10
11 def extract_structured_sections(filepath): usage
12     doc = fitz.open(filepath)
13     full_text = ""
14     page_map = {}
15     image_map = {}
16
17     for page_num, page in enumerate(doc, start=1):
18         text = page.get_text("text")
19         full_text += text + "\n"
20         page_map[page_num] = text
21
22         images = [
23             {"ext": img[1], "xref": img[0]}
24             for img in page.get_images(full=True)
25         ]
26         image_map[page_num] = images
27
28     doc.close()
29
30     sections = [s for s in re.split(pattern=r"(?=\n|\d+\.\d+)\s+", full_text) if s]
31
32     chunks = []
33
34     current_page = 1
35     seen_text = ""
36
37     def extract_structured_sections(filepath): usage
38         seen_text = ""
39
40         for i, section_text in enumerate(sections):
41             if not section_text or not isinstance(section_text, str):
42                 continue
43             cleaned = section_text.strip()
44
45             if len(cleaned) < 50:
46                 continue
47
48             # Estimate page number based on text offset
49             for page_num, page_text in page_map.items():
50                 if cleaned[:40] in page_text:
51                     current_page = page_num
52                     break
53
54             chunks.append({
55                 "text": cleaned,
56                 "page": current_page,
57                 "source": os.path.basename(filepath),
58                 "font_size": None, # Not available here
59                 "is_bold": None, # Not tracked here
60                 "type": "section",
61                 "images": image_map.get(current_page, []),
62                 "equations": [] # Placeholder
63             })
64
65     return chunks
66
67 def build_index(): usage
68     if os.path.exists(VECTOR_DB_PATH):
69         os.remove(VECTOR_DB_PATH)
70         print("Old vector index deleted.")
71
72     all_chunks, metadata = [], []
73
74     for file in os.listdir("data/raw"):
75         if not file.endswith(".pdf"):
76             continue
77
78         path = os.path.join("data/raw", file)
79         sections = extract_structured_sections(path)
80
81         for i, section in enumerate(sections):
82             all_chunks.append(section["text"])
83             metadata.append([
84                 "chunk_id": f"({file})-{section['page']}-{c(i+1)}",
85                 "page": section["page"],
86                 "source": section["source"],
87                 "type": section["type"],
88                 "font_size": section["font_size"],
89                 "is_bold": section["is_bold"],
90                 "images": section["images"],
91                 "equations": section["equations"]
92             })
93
94     vectors = embed_chunks(all_chunks)
95     vs = VectorStore(dim=vectors.shape[1])
96     vs.add(vectors, all_chunks, metadata)
97
98     os.makedirs(os.path.dirname(VECTOR_DB_PATH), exist_ok=True)
99     with open(VECTOR_DB_PATH, "wb") as f:
100         pickle.dump(vs, f)
101
102     print(f"Index built with {len(all_chunks)} chunks.")

```

Asked ChatGPT any major relevant enhancements I could make. It recommended sentence-based chunking with 3-5 sentence window overlap, citation filtering, and tagging section headers per chunk. I will start implementing those for better retrieval. I believe this will help improve accuracy majorly.

I moved all of preprocess\_pipeline to sentence-based chunking with window overlap, citation filtering, and tagged section headers. I ran into a bug with the NLTK download, but it was fixed. I added more citation and reference filtering to the bot as well as some abstract keyword boosting. I remade retriever.py to include keywords for introduction/abstract, but I realize this is only relevant to this 15 page example pdf. I will change it when I start implementing the 3000+ lecture notes. (This vector database now includes cos similarity to the query vector)

Retriever.py:

```
retriever.py | topic_indexer.py | DocSaver.py | app.py | preprocess_pipeline.py | extract_sent_chunks.py
```

```
import os
import pickle

from sklearn.metrics.pairwise import cosine_similarity as cos_sim

from ingest.preprocess_pipeline import build_index

VECTOR_DB_PATH = ".vector_index/index.pkl"

KEYWORDS = {"abstract", "introduction", "motivation", "problem", "approach", "transformer", "attention"}

def load_vectorstore():
    if not os.path.exists(VECTOR_DB_PATH):
        print(f"Vector DB path {VECTOR_DB_PATH} does not exist. Rebuilding it now...")
        build_index()

    if not os.path.exists(VECTOR_DB_PATH):
        raise FileNotFoundError(f"Missing index ({VECTOR_DB_PATH}) not found even after rebuild. Check data and pipeline.")

    with open(VECTOR_DB_PATH, "rb") as f:
        return pickle.load(f)

def cosine_similarity(a, b): 1 usage
    a = a.reshape(-1)
    b = b.reshape(-1)
    return cos_sim(a, b)[0][0]

def get_top_k_chunks(query_vec, vectorstore, k=5): 1 usage
    results = []
    for i, (vec, text, meta) in enumerate(zip(vectorstore.vectors, vectorstore.texts, vectorstore.metadata)):
        sim = cosine_similarity(query_vec, vec)

        # Denote citation/reference chunks
        if meta.get("type") == "citation" or meta.get("section", "").lower() == "references":
            sim *= 0.2

        # Boost chunks from abstract, intro, etc.
        section = meta.get("section", "").lower()
        if any(keyword in section for keyword in KEYWORDS):
            sim *= 1.4

        results.append((sim, text, meta))

    results.sort(key=lambda x: x[0], reverse=True)
    return results[:k]

def search(query_vec, vectorstore, k=5):
    return get_top_k_chunks(query_vec, vectorstore, k)

# Denote on import
retriever = load_vectorstore()
```

## Preprocess pipeline.py:

The screenshot shows a Jupyter Notebook interface with several open files. The current file is `preprocess_pipeline.py`, which contains Python code for document processing. Other visible files include `retriever.py`, `topic_indexer.py`, `DocSaver.py`, `app.py`, `preprocess_pipeline.py` (another instance), `extract_sent_chunks.py`, `extract_latex.py`, and `embedder.py`. The code in `preprocess_pipeline.py` includes imports for various libraries like PyTorch, NumPy, and re, along with custom classes and functions for reading documents, extracting sentences, and building index files.

```
retriever.py topic_indexer.py DocSaver.py app.py preprocess_pipeline.py extract_sent_chunks.py extract_latex.py embedder.py

# retriever.py
import os
import time
import PyPDF2
import pickle
import re
import sys
from collections import defaultdict
from nltk.tokenize import sent_tokenize
from embedder.embedder import Embedder
from embedder.vector_store import VectorStore

nltk.download('punkt')
VECTOR_DB_PATH = './vector_index/index.pkl'

SECTION_HEADER_PATTERN = re.compile(r"(?:(abstract|introduction|related works|methods|results|discussion|conclusion|references))", re.IGNORECASE)
CITATION_PATTERN = re.compile(r"\[(\w+\.\w+)\]\s*([^\]]*)")

def extract_structured_sections(filepath):
    doc = fitz.open(filepath)
    sections = []
    current_section = ''
    for page_num, page in enumerate(doc.get_pages()):
        blocks = page.get_text("dict").get("blocks")
        images = []
        for img in page.get_images(all=True):
            images.append(img)
        for block in blocks:
            if block["type"] == "text" and len(block["text"]) > 0:
                continue
            text = "\n".join(span["text"] for span in block["spans"] if span in block["lines"])
            if not text or len(text) < 10:
                continue
            text = " ".join(span["text"] for span in block["spans"] if span in line["spans"])
            if not text or len(text) < 10:
                continue
            if current_section != text:
                sections.append((text, current_section))
                current_section = text
            else:
                sections[-1][1] += f"\n{text}"
    doc.close()
    return sections

def build_index():
    if os.path.exists(VECTOR_DB_PATH):
        os.remove(VECTOR_DB_PATH)
        print("Old vector index deleted.")

# topic_indexer.py
# DocSaver.py
# app.py
# preprocess_pipeline.py
# extract_sent_chunks.py
# extract_latex.py
# embedder.py
```

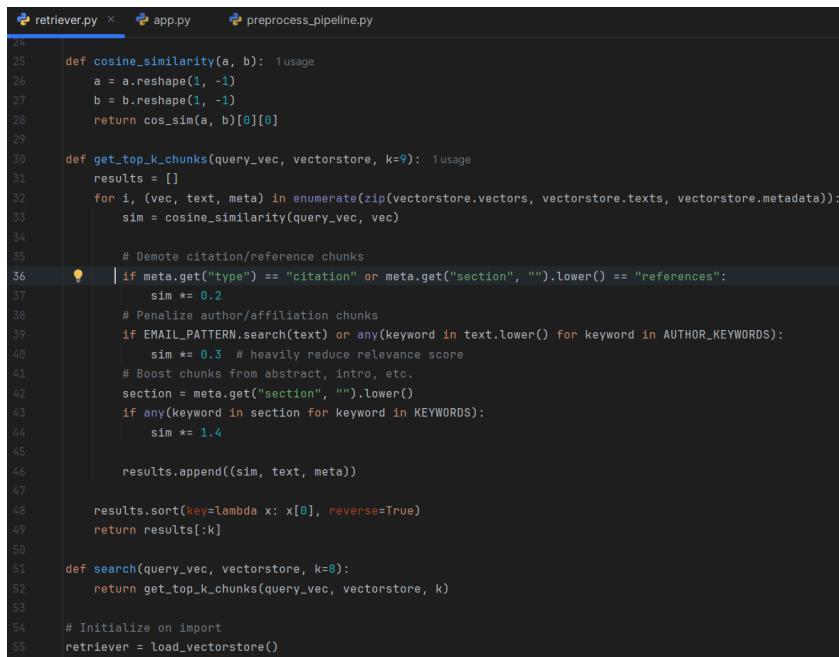
```
retriever.py  topic_indexer.py  DocSaver.py  app.py  preprocess_pipeline.py
 69 def build_index():
70     """Builds an index from raw PDF files in the 'data/raw' directory.
71     The index consists of chunks of text, their page numbers, and their source
72     files. Each chunk is associated with its type (e.g., text, images, equations)
73     and source section (e.g., 'Page', 'Source', 'Section')."""
74
75     all_chunks, metadata = [], []
76
77     for file in os.listdir("data/raw"):
78         if not file.endswith(".pdf"):
79             continue
80
81         print(f"Found PDF: {file}")
82         path = os.path.join("data/raw", file)
83
84         try:
85             sections = extract_structured_sections(path)
86         except Exception as e:
87             print(f"⚠️ Error processing {file}: {e}")
88             continue
89
90
91         path = os.path.join("data/raw", file)
92         sections = extract_structured_sections(path)
93
94         for i, section in enumerate(sections):
95             all_chunks.append(section["text"])
96             metadata.append({
97                 "chunk_id": f"{file}-{i}-{section['page']}",
98                 "page": section["page"],
99                 "source": section["source"],
100                 "section": section["section"],
101                 "type": section["type"],
102                 "images": section["images"],
103                 "equations": section["equations"]
104             })
105
106
107             vectors = embed_chunks(all_chunks)
108             vs = VectorStore(dim=vectors.shape[1])
109             vs.add(vectors, all_chunks, metadata)
110
111
112             os.makedirs(os.path.dirname(VECTOR_DB_PATH), exist_ok=True)
113             with open(VECTOR_DB_PATH, "wb") as f:
114                 pickle.dump(vs, f)
```

The number of chunks was moved up to 9 due to how small the chunks are now. The reference/citation filtering/nullifying seems to have helped a lot, but semantic relevance can still be improved. The relevance has definitely improved since the previous page-based chunking.

### Wednesday July 16 (7/16/25)

Goals: Mess with keyword boosting and citation filtering. Try using topic\_indexer potentially. Start integrating parts of the lecture notes to specialize the bot towards that style of literature, not strictly this scientific paper.

I started by adding some author and email keywords and adding a relevance penalizer to author names and emails.



```
retriever.py x app.py preprocess_pipeline.py
24
25     def cosine_similarity(a, b): 1usage
26         a = a.reshape(1, -1)
27         b = b.reshape(1, -1)
28         return cos_sim(a, b)[0][0]
29
30     def get_top_k_chunks(query_vec, vectorstore, k=9): 1usage
31         results = []
32         for i, (vec, text, meta) in enumerate(zip(vectorstore.vectors, vectorstore.texts, vectorstore.metadata)):
33             sim = cosine_similarity(query_vec, vec)
34
35             # Demote citation/reference chunks
36             if meta.get("type") == "citation" or meta.get("section", "").lower() == "references":
37                 sim *= 0.2
38             # Penalize author/affiliation chunks
39             if EMAIL_PATTERN.search(text) or any(keyword in text.lower() for keyword in AUTHOR_KEYWORDS):
40                 sim *= 0.3 # heavily reduce relevance score
41             # Boost chunks from abstract, intro, etc.
42             section = meta.get("section", "").lower()
43             if any(keyword in section for keyword in KEYWORDS):
44                 sim *= 1.4
45
46             results.append((sim, text, meta))
47
48         results.sort(key=lambda x: x[0], reverse=True)
49         return results[:k]
50
51     def search(query_vec, vectorstore, k=8):
52         return get_top_k_chunks(query_vec, vectorstore, k)
53
54     # Initialize on import
55     retriever = load_vectorstore()
```

I got the bot to detect author and email metadata using Regex and hard skip academic affiliation lines. I also filtered short, low-context lines as well as increased the window size to 5. I'm starting to think the sentence-based chunking won't work for the large lecture notes purely because they are too specific, but we'll see how it integrates.

Before integrating the first section of the lecture notes, I asked ChatGPT for recommendations on the chunking approach. It course corrected to a hybrid, section-sentence based chunking

approach. The hybrid:

1. Combines **section-based** and **sentence-window-based** chunking.
2. Skips short blocks and filters out author info/emails.
3. Uses overlapping sentence windows (5 sentences per chunk, 3-sentence stride).
4. Supports future TOC alignment.

I also implemented equation extracting to put equations into the sentence-section chunks. This led to a duplication of chunks error, so I put in a duplicate chunk skipper. I still have yet to fully implement the TOC indexer in retriever.py.

```

retriever.py app.py preprocess_pipeline.py
22 def extract_structured_sections(filepath): Usage
23     doc = fitz.open(filepath)
24     chunks = []
25     seen_texts = set() # Track duplicates
26     current_section = ""
27
28     for page_num, page in enumerate(doc, start=1):
29         blocks = page.get_text("dict")["blocks"]
30         images = [{"text": img[1], "xref": img[0]} for img in page.get_images()]
31
32         for block in blocks:
33             if "lines" not in block:
34                 continue
35
36             text = ".join(span["text"] for line in block["lines"] for span in line["spans"]).strip()
37             if not text or len(text) < 10:
38                 continue
39
40             # Detect new section heading
41             if SECTION_HEADER_PATTERN.match(text):
42                 current_section = text.strip()
43                 continue
44
45             # Sentence-level chunking
46             sentences = sent_tokenize(text)
47             window_size, step_size = 5, 3
48
49             for i in range(0, len(sentences), step_size):
50                 window = sentences[i:i + window_size]
51                 combined_text = " ".join(window)
52
53                 if len(combined_text.split()) < 10 or combined_text in seen_texts:
54                     continue # Skip duplicates or short chunks
55
56                 seen_texts.add(combined_text) # Mark as seen
57
58
59     def extract_structured_sections(filepath): Usage
60         seen_texts.add(combined_text) # Mark as seen
61
62         if EMAIL_PATTERN.search(combined_text.lower()):
63             continue # Skip author lines
64
65         is_citation = bool(CITATION_PATTERN.search(combined_text))
66         chunk_type = "citation" if is_citation or current_section.lower() == "references" else "paragraph"
67
68         # Find equations only in this sentence window
69         equations = []
70         for line in block.get("lines", []):
71             for span in line.get("spans", []):
72                 font = span.get("font", "").lower()
73                 if "symbol" in font or "math" in font:
74                     equations.append(span["text"])
75
76         # Math-heavy block text (optional enhancement)
77         if "=" in text and len(text) > 10:
78             equations.append(text.strip())
79
80         chunks.append({
81             "text": combined_text,
82             "page": page_num,
83             "source": os.path.basename(filepath),
84             "section": current_section,
85             "type": chunk_type,
86             "images": images,
87             "equations": equations
88         })
89
90     doc.close()
91
92     return chunks
93

```

I made the hybrid approach even more “hybrid” by having both section and sentence-based chunking. If the paragraphs are over 80 words (I can tweak it to be more or less words), it does paragraph/section-based chunking, and less than or equal to is sentence-based. I will mess around more with this. Here is an example of chunks pulled with this format:

```
[nltk_data] Downloading package punkt to
[nltk_data]   C:\Users\grant\AppData\Roaming\nltk_data...
[nltk_data] Package punkt is already up-to-date!
☒ Old vector index deleted.
```

Batches: 100% [██████████] 5/5 [00:01<00:00, 3.21it/s]

Index built with 139 chunks.

Choose output mode:

1. Display results
2. Save results to file (Downloads folder)

Enter 1 or 2: 1

Ask a question (or type 'exit'): What was the training section about?

Top Retrieved Chunks:

[1] From: TestPDFforRAGbot.pdf Page: 7 (Right section and right use of sentence chunking, but I wish it would include the formula given. I will tweak that more)

We varied the learning rate over the course of training, according to the formula:

---

[2] From: TestPDFforRAGbot.pdf Page: 8

The configuration of this model is listed in the bottom line of Table 3. Training took 3 . 5 days on 8 P100 GPUs. Even our base model surpasses all previously published models and ensembles, at a fraction of the training cost of any of the competitive models.

---

[3] From: TestPDFforRAGbot.pdf Page: 7 (These pulled from the right section, just not in chunking format. It makes me think I should switch back to only chunking format)

This corresponds to increasing the learning rate linearly for the first warmup \_ steps training steps, and decreasing it thereafter proportionally to the inverse square root of the step number. We used warmup \_ steps = 4000 .

---

[4] From: TestPDFforRAGbot.pdf Page: 7

Sentence pairs were batched together by approximate sequence length. Each training batch contained a set of sentence pairs containing approximately 25000 source tokens and 25000 target tokens.

---

[5] From: TestPDFforRAGbot.pdf Page: 11 (Still need to do more similarity punishing for citations)

[9] Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N. Dauphin. Convolutional sequence to sequence learning. arXiv preprint arXiv:1705.03122v2 , 2017.

---

[6] From: TestPDFforRAGbot.pdf Page: 7 (This one pulled the correct paragraph!) :D

We trained our models on one machine with 8 NVIDIA P100 GPUs. For our base models using the hyperparameters described throughout the paper, each training step took about 0.4 seconds. We trained the base models for a total of 100,000 steps or 12 hours. For our big models,(described on the bottom line of table 3), step time was 1.0 seconds. The big models were trained for 300,000 steps (3.5 days).

---

[7] From: TestPDFforRAGbot.pdf Page: 9 (This one pulled from the wrong page/section, but it is changeable)

We trained a 4-layer transformer with d model = 1024 on the Wall Street Journal (WSJ) portion of the Penn Treebank [ 25 ], about 40K training sentences. We also trained it in a semi-supervised setting, using the larger high-confidence and BerkleyParser corpora from with approximately 17M sentences [ 37 ]. We used a vocabulary of 16K tokens for the WSJ only setting and a vocabulary of 32K tokens for the semi-supervised setting.

---

[8] From: TestPDFforRAGbot.pdf Page: 6

We also experimented with using learned positional embeddings [ 9 ] instead, and found that the two versions produced nearly identical results (see Table 3 row (E)). We chose the sinusoidal version because it may allow the model to extrapolate to sequence lengths longer than the ones encountered during training.

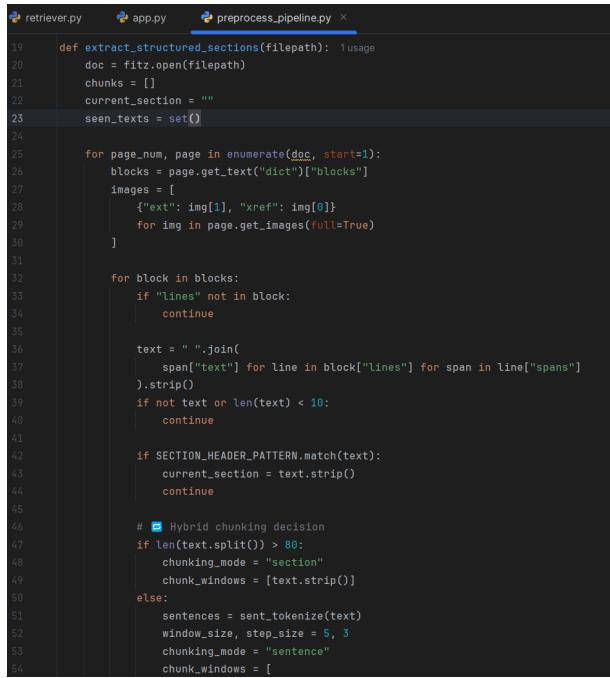
---

[9] From: TestPDFforRAGbot.pdf Page: 11

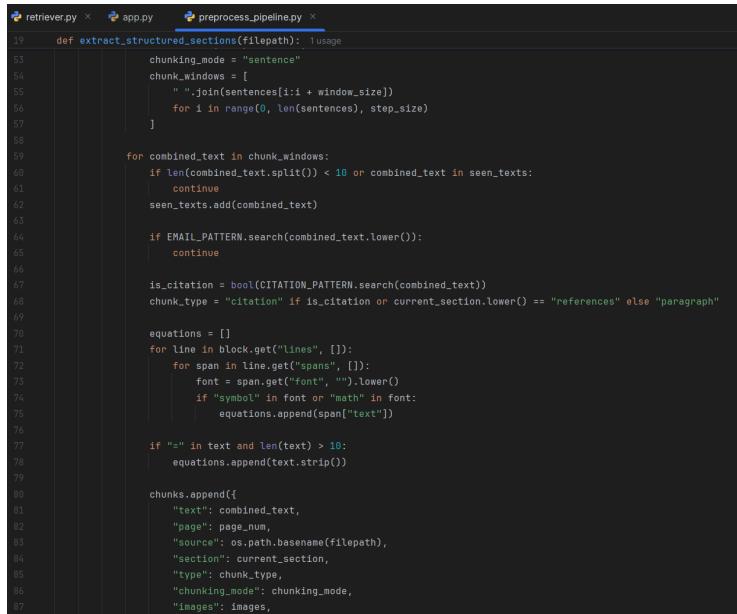
[19] Yoon Kim, Carl Denton, Luong Hoang, and Alexander M. Rush. Structured attention networks. In International Conference on Learning Representations , 2017.

---

Ask a question (or type 'exit'): exit



```
retriever.py app.py preprocess_pipeline.py
19 def extract_structured_sections(filepath): 1usage
20     doc = fitz.open(filepath)
21     chunks = []
22     current_section = ""
23     seen_texts = set()
24
25     for page_num, page in enumerate(doc, start=1):
26         blocks = page.get_text("dict")["blocks"]
27         images = [
28             {"text": img[1], "xref": img[0]}
29             for img in page.get_images(full=True)
30         ]
31
32         for block in blocks:
33             if "lines" not in block:
34                 continue
35
36             text = ".join(
37                 span["text"] for line in block["lines"] for span in line["spans"]
38             ).strip()
39             if not text or len(text) < 10:
40                 continue
41
42             if SECTION_HEADER_PATTERN.match(text):
43                 current_section = text.strip()
44                 continue
45
46             # ↗ Hybrid chunking decision
47             if len(text.split()) > 80:
48                 chunking_mode = "section"
49                 chunk_windows = [text.strip()]
50             else:
51                 sentences = sent_tokenize(text)
52                 window_size, step_size = 5, 3
53                 chunking_mode = "sentence"
54                 chunk_windows = [
55
```



```
retriever.py app.py preprocess_pipeline.py
19 def extract_structured_sections(filepath): 1usage
20     chunking_mode = "sentence"
21     chunk_windows = [
22         ".join(sentences[i:i + window_size])
23         for i in range(0, len(sentences), step_size)
24     ]
25
26     for combined_text in chunk_windows:
27         if len(combined_text.split()) < 10 or combined_text in seen_texts:
28             continue
29         seen_texts.add(combined_text)
30
31         if EMAIL_PATTERN.search(combined_text.lower()):
32             continue
33
34         is_citation = bool(CITATION_PATTERN.search(combined_text))
35         chunk_type = "citation" if is_citation or current_section.lower() == "references" else "paragraph"
36
37         equations = []
38         for line in block.get("lines", []):
39             for span in line.get("spans", []):
40                 font = span.get("font", "").lower()
41                 if "symbol" in font or "math" in font:
42                     equations.append(span["text"])
43
44         if "=" in text and len(text) > 10:
45             equations.append(text.strip())
46
47         chunks.append({
48             "text": combined_text,
49             "page": page_num,
50             "source": os.path.basename(filepath),
51             "section": current_section,
52             "type": chunk_type,
53             "chunking_mode": chunking_mode,
54             "images": images,
```

I added a crude TOC parser as a file in ingest. This will help assign semantic values to sections to increase their similarity to the query when prompted. I also adjusted preprocess\_pipeline to accommodate this. I will have to change it for the lecture notes as the TOC is only in the first of the eleven pdfs.

```
1 # toc_parser.py
2
3 import fitz # PyMuPDF
4 import re
5
6 TOC_SECTION_PATTERN = re.compile(pattern=r"^(chapter|section|part)?\s*\d+(\.\d+)*(\s+.+)?$", re.IGNORECASE)
7
8 def parse_toc(filepath, max_pages=89): 2 usages
9     doc = fitz.open(filepath)
10    toc_sections = []
11
12    for page_num in range(min(max_pages, len(doc))):
13        page = doc[page_num]
14        blocks = page.get_text("dict")["blocks"]
15
16        for block in blocks:
17            if "lines" not in block:
18                continue
19            text = ".join(span["text"] for line in block["lines"] for span in line["spans"]).strip()
20            if TOC_SECTION_PATTERN.match(text):
21                toc_sections.append(text)
22
23    doc.close()
24    return toc_sections
25
```

### Thursday July 17th (7/17/25)

Goals: Implement the first section of the Lecture Notes PDF. Add the full TOC parser for that first section. Tweak more with the formulas and citation preferences. Keep trying to make the chunks more accurate.

I'm going to start slowly. I won't implement the full TOC parser just yet, but I will add the first section to see how the preprocess pipeline reacts.

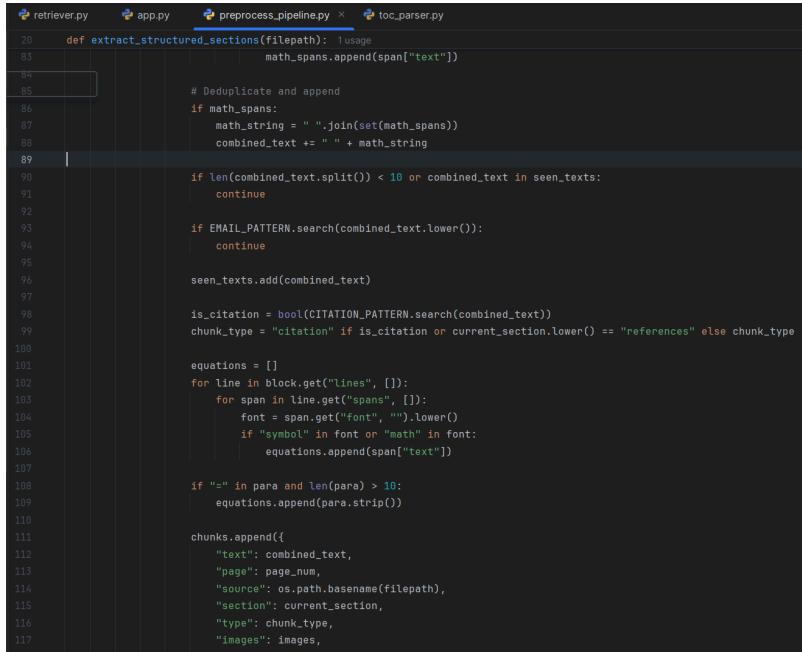
Before starting, I asked ChatGPT for any final recommendations of what to add. It recommended using style header recognition, paragraph merging, and figure/table caption preservation. I added those all to the code. I also asked ChatGPT about page-chunking again due to how long each of the subsections are. It agreed it was wise to make an ultra-hybrid model, consisting of small, medium, and page sized-chunks. I added the ultra-hybrid model as well as better equation and formula handling (It will now include it in the chunks, hopefully).

```
retriever.py app.py preprocess_pipeline.py toc_parser.py

16 EMAIL_PATTERN = re.compile(r"([a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+\.[a-zA-Z]{2,})")
17 FIGURE_PATTERN = re.compile(r"(figure|table)\s*(\d+(\.\d+)*", re.IGNORECASE)
18
19
20 def extract_structured_sections(filepath):  usage
21     doc = fitz.open(filepath)
22     chunks = []
23     current_section = ""
24     seen_texts = set()
25
26     for page_num, page in enumerate(doc, start=1):
27         blocks = page.get_text("dict")["blocks"]
28         images = [
29             {"ext": img[1], "ref": img[0]}
30             for img in page.get_images(full=True)
31         ]
32
33         for block in blocks:
34             if "lines" not in block:
35                 continue
36
37             lines = [" ".join(span["text"] for span in line["spans"].strip() for line in block["lines"])]
38             paragraphs = []
39             current_para = ""
40
41             # Merge broken lines into paragraphs
42             for line in lines:
43                 if not line:
44                     continue
45                 if line[-1] in ".;?!":
46                     current_para += " " + line
47                     paragraphs.append(current_para.strip())
48                     current_para = ""
49                 else:
50                     current_para += " " + line
51
52             if current_para:
```

```
retriever.py app.py preprocess_pipeline.py toc_parser.py

20     def extract_structured_sections(filepath):
21         """Usage
22             current para += " " + line
23             if current_pera:
24                 paragraphs.append(current_pera.strip())
25
26             for para in paragraphs:
27                 if not para or len(para) < 10:
28                     continue
29
30                     # Detect new section heading
31                     if SECTION_HEADER_PATTERN.match(para):
32                         current_section = para.strip()
33                         continue
34
35                     # Detect and store figure/table captions
36                     if FIGURE_PATTERN.match(para):
37                         chunk_type = "caption"
38                     else:
39                         chunk_type = "paragraph"
40
41                     # Sentence chunking
42                     sentences = sent_tokenize(para)
43                     window_size, step_size = 5, 3
44                     for i in range(0, len(sentences), step_size):
45                         window = sentences[i:i + window_size]
46                         combined_text = " ".join(window)
47
48                         # Find math spans in the block and append them to the chunk if they weren't already included
49                         math_spans = []
50                         for line in block.get("Lines", []):
51                             for span in line.get("spans", []):
52                                 font = span.get("font", "").lower()
53                                 if "symbol" in font or "math" in font or re.search(r"[\d\w]+\s*=\s*[\d\w]+", span["text"]): # crude math check
54                                     math_spans.append(span["text"])
55
56                         combined_text += " " + " ".join(math_spans)
57
58                         block["Lines"] = [{"text": combined_text}]]
```



```
retriever.py app.py preprocess_pipeline.py toc_parser.py
20     def extract_structured_sections(filepath):  usage
21         with open(filepath, "r") as f:
22             file_content = f.read()
23
24         # Deduplicate and append
25         if math_spans:
26             math_string = " ".join(set(math_spans))
27             combined_text += " " + math_string
28
29         if len(combined_text.split()) < 10 or combined_text in seen_texts:
30             continue
31
32         if EMAIL_PATTERN.search(combined_text.lower()):
33             continue
34
35         seen_texts.add(combined_text)
36
37         is_citation = bool(CITATION_PATTERN.search(combined_text))
38         chunk_type = "citation" if is_citation or current_section.lower() == "references" else chunk_type
39
40         equations = []
41         for line in block.get("lines", []):
42             for span in line.get("spans", []):
43                 font = span.get("font", "").lower()
44                 if "symbol" in font or "math" in font:
45                     equations.append(span["text"])
46
47         if "=" in para and len(para) > 10:
48             equations.append(para.strip())
49
50         chunks.append({
51             "text": combined_text,
52             "page": page_num,
53             "source": os.path.basename(filepath),
54             "section": current_section,
55             "type": chunk_type,
56             "images": images,
57         })
58
59     return chunks
```

Not having implemented the TOC parser yet, I tested these new improvements with the first of the eleven PDFs. The chunks were all cut off; I will try to fix it.

After a long time of trying new things: I decided to ditch the sentence-chunking for section and page-chunking only. I also disabled sentence-window chunking for long text. I included merging blocks by page and mode metadata to support chunk-type tracking. Even after adding all of that, the chunks were still sentences that were cut off. Finally ChatGPT said the only way to help would be to use a TOC parser which I finally implemented. I had trouble with the formatting of the TOC lines (The parser couldn't read it due to how PyMuPDF works), so I'm working around that to make the code work for the formatting.

(I also ran into issues with unknown conversation limitations. After such a long conversation on the plus account I was not able to ask any more queries. I googled it and it is supposedly due to high traffic times during the day and a maximum amount of queries within 3 hours. Needless to say I made a different conversation and updated it on the other conversation I had left).

```
retriever.py app.py preprocess_pipeline.py toc_parser.py ×  
1 import fitz  
2 import re  
3  
4 TOC_ROW_PATTERN = re.compile(r"\d+(\.\d+)+\s+(.+?)\s+(\d+)")  
5  
6 def parse_toc(filepath, max_pages=89): 2 usages  
7     doc = fitz.open(filepath)  
8     toc = []  
9     candidate_lines = []  
10  
11     for page in doc[:max_pages]:  
12         lines = page.get_text("text").split("\n")  
13         for line in lines:  
14             candidate_lines.append(line.strip())  
15  
16     # Merge every 2-3 lines to reassemble TOC rows  
17     for i in range(len(candidate_lines) - 2):  
18         merged = f"{candidate_lines[i]} {candidate_lines[i+1]} {candidate_lines[i+2]}"  
19         match = TOC_ROW_PATTERN.match(merged)  
20         if match:  
21             section = match.group(1)  
22             title = match.group(3).strip()  
23             page_num = int(match.group(4))  
24             toc.append({  
25                 "section": section,  
26                 "title": title,  
27                 "page": page_num  
28             })  
29  
30     doc.close()  
31     return toc
```

```
retriever.py app.py preprocess_pipeline.py toc_parser.py  
19  
20  
21 def extract_structured_sections(filepath, toc_lookup=None): 1 usage  
22     doc = fitz.open(filepath)  
23     chunks = []  
24     seen_texts = set()  
25     current_section = "Unknown Section"  
26  
27     section_by_page = {}  
28     if toc_lookup:  
29         sorted_toc = sorted(toc_lookup, key=lambda x: x["page"])  
30         for i, entry in enumerate(sorted_toc):  
31             start = entry["page"]  
32             end = sorted_toc[i + 1]["page"] if i + 1 < len(sorted_toc) else 10_000  
33             for p in range(start, end):  
34                 section_by_page[p] = entry["title"]  
35  
36     for page_num, page in enumerate(doc, start=1):  
37         current_section = section_by_page.get(page_num, current_section)  
38  
39         blocks = page.get_text("dict")["blocks"]  
40         images = [{"ext": img[1], "xref": img[0]} for img in page.get_images(full=True)]  
41  
42         page_paragraphs = []  
43         for block in blocks:  
44             if "lines" not in block:  
45                 continue  
46  
47             text = ".join(span["text"] for line in block["lines"] for span in line["spans"]).strip()  
48             if not text or len(text) < 10:  
49                 continue  
50  
51             # Section header detection (fallback if TOC not used)  
52             if SECTION_HEADER_PATTERN.match(text):  
53                 current_section = text.strip()  
54                 continue
```

```
retriever.py app.py preprocess_pipeline.py toc_parser.py
21 def extract_structured_sections(filepath, toc_lookup=None): 1 usage
56     if EMAIL_PATTERN.search(text.lower()):
57         continue
58
59     page_paragraphs.append(text)
60
61     # Group paragraphs by page
62     grouped = []
63     current_group = []
64     word_count = 0
65
66     for para in page_paragraphs:
67         wc = len(para.split())
68         if para in seen_texts or wc < 10:
69             continue
70         seen_texts.add(para)
71
72         current_group.append(para)
73         word_count += wc
74
75         # If group size exceeds ~300 words, commit
76         if word_count >= 300:
77             grouped.append(" ".join(current_group))
78             current_group = []
79             word_count = 0
80
81     # Add remainder
82     if current_group:
83         grouped.append(" ".join(current_group))
84
85     # Process each grouped chunk
86     for para_group in grouped:
87         eqs = []
88         if "=" in para_group or any(w in para_group for w in ["\%", "\&", "\>", "\f", "\Sigma", "\pi", "\lambda"]):
89             eqs.append(para_group) # crude math heuristic
```

```
retriever.py app.py preprocess_pipeline.py toc_parser.py
21 def extract_structured_sections(filepath, toc_lookup=None): 1 usage
89     eqs.append(para_group) # crude math heuristic
90
91     chunks.append({
92         "text": para_group,
93         "page": page_num,
94         "source": os.path.basename(filepath),
95         "section": current_section,
96         "type": "paragraph-group",
97         "images": images,
98         "equations": eqs,
99         "mode": "page-paragraph"
100    })
101
102    doc.close()
103    return chunks
104
105 TOC_GLOBAL = None # Declare global variable for reuse
106
107    def build_index(): 4 usages
108        if os.path.exists(VECTOR_DB_PATH):
109            os.remove(VECTOR_DB_PATH)
110            print("Old vector index deleted.")
111
112        all_chunks, metadata = [], []
113        toc_titles = None
114
115        for i, file in enumerate(sorted(os.listdir("data/raw"))):
116            if not file.endswith(".pdf"):
117                continue
118
119            path = os.path.join("data/raw", file)
120
121            # Parse TOC only once
122            if toc_titles is None and "LectureNotes-1.pdf" in file:
123                toc_titles = parse_toc(path)
```

```

107     def build_index(): 4 usages
108
109         path = os.path.join("data/raw", file)
110
111         # Parse TOC only once
112         if toc_titles is None and "LectureNotes-1.pdf" in file:
113             toc_titles = parse_toc(path)
114             with open(TOC_CACHE_PATH, "wb") as f:
115                 pickle.dump(toc_titles, f)
116             print(f"\u2b1c TOC parsed with {len(toc_titles)} entries.")
117         elif toc_titles is None:
118             with open(TOC_CACHE_PATH, "rb") as f:
119                 toc_titles = pickle.load(f)
120
121         try:
122             sections = extract_structured_sections(path, toc_titles)
123         except Exception as e:
124             print(f"\u2b1a Error processing {file}: {e}")
125             continue
126
127         for i, section in enumerate(sections):
128             all_chunks.append(section["text"])
129             metadata.append({
130                 "chunk_id": f"{file}-p{section['page']}-c{i + 1}",
131                 "page": section["page"],
132                 "source": section["source"],
133                 "section": section["section"],
134                 "type": section["type"],
135                 "images": section["images"],
136                 "equations": section["equations"],
137                 "mode": section["mode"]
138             })
139
140         if not all_chunks:
141             print("\u26a0\ufe0f No valid chunks extracted. Index not created.")
142             return
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163

```

```

107     def build_index(): 4 usages
108
109
110
111
112
113
114
115
116 More tool windows
117     for i, section in enumerate(sections):
118         all_chunks.append(section["text"])
119         metadata.append({
120             "chunk_id": f"{file}-p{section['page']}-c{i + 1}",
121             "page": section["page"],
122             "source": section["source"],
123             "section": section["section"],
124             "type": section["type"],
125             "images": section["images"],
126             "equations": section["equations"],
127             "mode": section["mode"]
128         })
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163

```

I got it to work for 50 or so of the lines, however it still needs fixing. To fix the problem I added block retrieval instead of only retrieving text. This led to over 7000+ titles being parsed, which isn't ideal. I then decided to add multiple tools to lower this number. This included:

- Block-based extraction (handles missing `text()` content)
- Smart sliding window parsing (handles broken TOC lines)
- Depth tracking based on section number (e.g. `102.4.1` = depth 3)
- Outputs each TOC entry as a `{"title", "page", "depth"}` dictionary
- Passes those TOC entries to `extract_structured_sections()`

- Caches TOC to disk for reuse across multi-part files
- Adds better error handling and consistent logging

As well as later when this made too many parses again (1700+), I implemented filters to lower it. This included limiting the parsing to pages 14 to 89 (it was 1-89 but I realized the TOC starts on the 14th page), and filtering out header and footer lines. This led to it being too strict and no titles were being parsed, so I tweaked it until 1700+ titles were parsed. Here are some example debug lines that I will use to work on tomorrow:

```
[DEBUG] Page 87: Elastic Beam, 27 Node Brick Model With Concentrated Mass.
[DEBUG] Page 87: 3004 (This needs to be on the line above for page number)
[DEBUG] Page 87: Jeremić et al.
[DEBUG] Page 87: University of California, Davis (These headers and footers to be filtered out)
[DEBUG] Page 87: version: 28Apr2025, 17:07
[DEBUG] Page 88: Jeremić et al., Real-ESSI
[DEBUG] Page 88: ESSI Notes (These side texts need to be filtered out)
[DEBUG] Page 88: CONTENTS
[DEBUG] Page 88: page: 88 of 3286
[DEBUG] Page 88: 707.8
[DEBUG] Page 88: Elastic Beam Element, Dynamic Loading, Viscous (Rayleigh/Caughey) and
Nu-
[DEBUG] Page 88: merical (Newmark/HHT) Damping .
[DEBUG] Page 88: 3011
[DEBUG] Page 88: 707.9 (This needs to be all in one line as with similar lines)
[DEBUG] Page 88: Elastic Beam Element for a Simple Frame Structure
[DEBUG] Page 88: 3018
[DEBUG] Page 88: 707.10 27NodeBrick Cantilever Beam, Static Load .
[DEBUG] Page 88: 3020
```

```

retriever.py app.py preprocess_pipeline.py toc_parser.py

4 SECTION_PATTERN = re.compile(r"\d+(\.\d+)*$")
5 PAGE_PATTERN = re.compile(r"\d{1,4}$")
6
7 def extract_lines_from_blocks(filepath, min_page=13, max_page=89):
8     """Extracts TOC-relevant lines from pages 13 to 89, skipping headers/footers."""
9     doc = fitz.open(filepath)
10    lines = []
11
12    for page_num in range(min_page, max_page): # page 13 to 89
13        page = doc[page_num]
14        for block in page.get_text("blocks"):
15            if len(block) >= 5:
16                x0, y0, x1, y1, text = block[:5]
17
18            if y0 < 50 or y1 > 780:
19                continue
20
21            for line in text.split("\n"):
22                cleaned = line.strip()
23                if cleaned and cleaned.strip(".") != "":
24                    print(f"[DEBUG] Page {page_num + 1}: {cleaned}") # Optional: confirm line
25                    lines.append(cleaned)
26
27    doc.close()
28    return lines
29
30 def parse_toc(filepath, max_pages=89):
31     """
32         Parses the table of contents from the first `max_pages` of a PDF,
33         returning a list of dicts: {title, page, depth}
34     """
35     TOC_START_PAGE = 13 # page 13 (0-indexed)
36     TOC_END_PAGE = 89
37     lines = extract_lines_from_blocks(filepath, min_page=TOC_START_PAGE, max_page=TOC_END_PAGE)
38     toc = []
39     buffer = []

```

```

retriever.py app.py preprocess_pipeline.py toc_parser.py

30 def parse_toc(filepath, max_pages=89):
31     for line in lines:
32         buffer.append(line)
33
34         # Try all 3-line windows in the buffer
35         if len(buffer) >= 3:
36             for i in range(len(buffer) - 2):
37                 a, b, c = buffer[i], buffer[i+1], buffer[i+2]
38
39             if SECTION_PATTERN.match(a) and PAGE_PATTERN.match(c):
40                 try:
41                     section = a
42                     title = b.rstrip(".") # Remove trailing dot
43                     page = int(c)
44                     depth = section.count(".") + 1
45
46                     toc.append({
47                         "title": f"{section} {title}",
48                         "page": page,
49                         "depth": depth
50                     })
51
52                     buffer = buffer[i+3:] # Remove the matched block
53                     break
54
55                 except Exception as e:
56                     print(f"⚠️ Skipping malformed TOC entry: {a}, {b}, {c} - {e}")
57
58             else:
59                 # Trim runaway buffer
60                 if len(buffer) > 10:
61                     buffer.pop(0)
62
63
64     print(f"💡 TOC parsed with {len(toc)} entries.")
65     if toc:
66         print(f"✅ Sample TOC entry: {toc[0]}")
67
68
69
70
71
72
73
74

```

Friday July 18th (7/18/25)

Goals: Continue tweaking TOC indexer to get accurate parsed titles. This may include making a better smart sliding window or block-based extraction. If successful: add more of the lecture

notes PDFs (Currently only using the first one). If that integrates well: start performing validation tests to see how accurate the chunks and ChatGPT responses are.

I'll start with yesterday's final ChatGPT recommendations which means putting in a bad line pattern set to skip header/footer lines with key words.

I implemented the bad line pattern sets so that the parser filters out anything with those key words, and I updated the parsing logic to accommodate for it. This let some of the titles be filtered out from the key words. Some longer titles that cover multiple lines were also messing up the parsing. It was recommended to fix this through a smart buffer assembly which I implemented. It does the following:

Refactor your parsing loop to:

- Detect a **SECTION\_PATTERN** line
- Collect all following lines until a valid **PAGE\_PATTERN** is found
- Treat *everything between* as part of the title
- Multi-line titles (any number of lines)
- Embedded date/version lines (like **(2010-2012...)**) as part of the title
- Robust fallback if structure is malformed

After this, I worked out that some of the titles were still being filtered out, so that is what I will work on right now. I also read through an article on making a RAG system for PDFs. There was some good advice on what I could do to improve, but mainly it covered the basics. It may be a good example of how to structure our paper.

Article: Developing Retrieval Augmented Generation (RAG) based LLM Systems from PDFs:  
An Experience Report

```

43     return lines
44
45     def parse_toc(filepath, min_page=13, max_page=89): 2 usages
46         """
47             Parses the table of contents from a PDF, using robust multi-line handling.
48             Returns a list of dicts: {title, page, depth}
49         """
50
51         lines = extract_lines_from_blocks(filepath, min_page=min_page, max_page=max_page)
52         toc = []
53         i = 0
54
55         while i < len(lines) - 2:
56             section_candidate = lines[i]
57             title_lines = []
58             page_candidate = None
59
60             if SECTION_PATTERN.match(section_candidate):
61                 j = i + 1
62
63                 # Collect title lines until we hit a page number
64                 while j < len(lines):
65                     if PAGE_PATTERN.match(lines[j]):
66                         page_candidate = int(lines[j])
67                         break
68                     else:
69                         title_lines.append(lines[j])
70                     j += 1
71
72             if page_candidate and title_lines:
73                 title_text = " ".join(title_lines).strip().rstrip(".")
74                 depth = section_candidate.count(".") + 1
75                 toc.append({
76                     "title": f"{section_candidate} {title_text}",
77                     "page": page_candidate,
78                     "depth": depth
79                 })
80                 i = j + 1 # move past the page number
81             else:
82                 i += 1 # skip to next line if malformed
83             else:
84                 i += 1
85
86             print(f"\n\t TOC parsed with {len(toc)} entries.")
87             if toc:
88                 print(f"\n\t\t Sample TOC entry: {toc[0]}")
89
90         return toc

```

After barely tweaking the bad line pattern set, the parsed correctly and filtered out only the unnecessary info. It still is parsing 2052 TOC entries, which is wrong. It should be closer to possibly a few hundred. I will check for, and filter out, and duplicates.

After doing the math and realizing that the number of entries might be actually closer to 2000+, I realized that 2052 entries sounds about right. It turns out that after implementing the duplicate checker, it was page numbers being stuck in titles that were being detected as duplicates. I also added more multi-line title allowance to up to 4 lines of a title (The max I've seen in the TOC). I also updated the toc\_parser to get rid of the page numbers accidentally getting added to the beginning of titles. I added tons of debug logging, smart removal of page numbers in titles, duplicate tracking and filtering, but nothing worked.

After trying to fix this logistical bug for the entirety of the rest of the day, Aditya recommended I reformat the TOC because of how poorly formatted it is. I emailed Dr. Jeremic for the .tex file because that would make it a lot easier, but he hasn't responded. I will use ChatGPT to reformat the TOC to make it a lot easier to make the parser.

Friday July 19th 7/19/25

I spent the entire day manually getting the TOC parsed/reformatted to get the parser to work. So far, no good. I'm planning on cutting it and working/fine-tuning what I have left. If I get the .tex file from Dr. Jeremic, I may try again to work on it.

## Tuesday July 22nd 7/22/25

This morning I worked on the first part of the methodology. I described how RAG works. Now I'm going to begin to explain how my methodology went. Close to 2 or 2:30 I will try and see how the .tex file might work for the TOC parser. Dr. Jeremic finally emailed it to me, so I'll try again one last time. I was very close to finishing it, so I'm hoping this could be the last piece of the puzzle.

I started the second part of my methodology. I made an outline of what I want to include as well as key details that are important. I will start and hopefully finish writing it tomorrow.

## Wednesday July 23rd 2025 7/23/25

Goals: Read up on best practices for retrieval in RAG, implement new techniques to increase accuracy, test improved accuracy, continue and/or finish the methodology section.

Today I plan on stopping the toc parser and solely focusing on increasing the chunking accuracy of the retrieval bot. I spent most of the morning skimming an article solely focused on best practices for making a RAG framework for LLMs as well as recommendations from perplexity ai or ChatGPT of what to improve. I'm going to use that scientific paper as the basis for improvement instead of ChatGPT, but I'll use ChatGPT to help implement some of the best practices. I plan on continuing or finishing the methodology of the rough draft today as well.

It turns out Dr. Jeremic sent the .toc file, so I'll be working on that. It'll implement some more metadata and better chunking, so it works with some of the best practices.

After a while of fixing the parser, the TOC was finally able to be parsed. I rewrote the code for preprocess pipeline to help with section-stitching which hopefully increased chunk-query similarity.

toc\_parser.py:

```
retriever.py  toc_parser.py  CompMechanicsLectureNotesTOC.toc  app.py  preprocess_pipeline.py
1 import re
2
3 def parse_toc_file(toc_path): 3 usages
4     with open(toc_path, 'r', encoding='utf-8') as file:
5         lines = file.readlines()
6
7     entries = []
8     for raw_line in lines:
9         line = str(raw_line).strip() # Ensure it's a string
10
11         # Only process lines that start with \contentsline
12         if not line.startswith(r'\contentsline'):
13             continue
14
15         # Match chapter/section lines
16         match = re.search(
17             pattern: r'`\|\contentsline{[s*]{(chapter|section|subsection|subsubsection)}\|`'
18             r'|`{1}{s*}{\numberline{((^| ))}{s*}{(^| )}}`'
19             r'|`{1}{(d| )}`', line)
20
21     if not match:
22         continue
23
24     section_type, number, title, page = match.groups()
25
26     # Extract the correct page number just before (chapter.xxx)
27     page_match = re.search(pattern: r'{1}{(d| )}{1}{(?:chapter|section|subsection|subsubsection)}{1}{.^*}', line)
28
29     if not page_match:
30         continue
31     page = int(page_match.group(1))
32
33     # Clean title
34     title = re.sub(pattern: r'`{1}{[a-zA-Z]*}{s*}', repl: '', title) # remove LaTeX commands like \relax
35     title = re.sub(pattern: r'{1}{[^{}]*}', repl: '', title) # remove {...}
36     title = re.sub(pattern: r'{1}{<[23]>}{b}{d}{1}{b}{?}{0}{1}', repl: '', title) # Preserve "20" and "30" patterns, remove other stray numbers
37     title = re.sub(pattern: r'{1}{\{`}', repl: '', title) # remove stray \ or braces
38
39     entries.append({
40         'number': number.strip(),
41         'title': title,
42         'page': int(page)
43     })
44
45     return entries
46
47 # Example usage (uncomment for standalone test):
48 # parsed = parse_toc_file("CompMechanicsLectureNotesTOC.toc")
49 # for entry in parsed[:10]:
50 #     print(entry)
51
52 # Optional test run
53 if __name__ == "__main__":
54     toc_file = "./data/CompMechanicsLectureNotesTOC.toc"
55     toc_entries = parse_toc_file(toc_file)
56     for e in toc_entries[:100]: # Display first 20 entries for inspection
57         print(e)
```

## Preprocess\_pipeline.py:

```
retriever.py  toc_parser.py  CompMechanicsLectureNotesTOC.toc  app.py  preprocess_pipeline.py ×  
16  
17  
18     def extract_structured_sections(filepath, toc_lookup=None):  1 usage  
19         doc = fitz.open(filepath)  
20         chunks = []  
21         seen_texts = set()  
22         current_section = "Unknown Section"  
23  
24         section_by_page = {}  
25         if toc_lookup:  
26             sorted_toc = sorted(toc_lookup, key=lambda x: x["page"])  
27             for i, entry in enumerate(sorted_toc):  
28                 start = entry["page"]  
29                 end = sorted_toc[i + 1]["page"] if i + 1 < len(sorted_toc) else 10_000  
30                 for p in range(start, end):  
31                     section_by_page[p] = entry["title"]  
32  
33             for page_num, page in enumerate(doc, start=1):  
34                 current_section = section_by_page.get(page_num, current_section)  
35  
36                 blocks = page.get_text("dict")["blocks"]  
37                 images = [{"ext": img[1], "xref": img[0]} for img in page.get_images(full=True)]  
38  
39                 page_paragraphs = []  
40                 for block in blocks:  
41                     if "lines" not in block:  
42                         continue  
43  
44                     text = " ".join(span["text"] for line in block["lines"] for span in line["spans"]).strip()  
45                     if not text or len(text) < 10:  
46                         continue  
47  
48                     if SECTION_HEADER_PATTERN.match(text):  
49                         current_section = text.strip()  
50                         continue
```

```
retriever.py  toc_parser.py  CompMechanicsLectureNotesTOC.toc  app.py  preprocess_pipeline.py ×  
18     def extract_structured_sections(filepath, toc_lookup=None):  1 usage  
50         continue  
51  
52         if EMAIL_PATTERN.search(text.lower()):  
53             continue  
54  
55             page_paragraphs.append(text)  
56  
57             grouped = []  
58             current_group = []  
59             word_count = 0  
60  
61             for para in page_paragraphs:  
62                 wc = len(para.split())  
63                 if para in seen_texts or wc < 10:  
64                     continue  
65                 seen_texts.add(para)  
66  
67                 current_group.append(para)  
68                 word_count += wc  
69  
70                 if word_count >= 300:  
71                     grouped.append(" ".join(current_group))  
72                     current_group = []  
73                     word_count = 0  
74  
75                 if current_group:  
76                     grouped.append(" ".join(current_group))  
77  
78             for para_group in grouped:  
79                 eqs = []  
80                 if "=" in para_group or any(w in para_group for w in ["\u2203", "\u03b4", "\u2192", "\u221f", "\u2211", "\u03c0", "\u03bb"]):  
81                     eqs.append(para_group)  
82  
83                     chunks.append({  
84                         "text": para_group,
```

The screenshot shows a code editor interface with several tabs at the top: retriever.py, toc\_parser.py, CompMechanicsLectureNotesTOC.toc, app.py, and preprocess\_pipeline.py. The preprocess\_pipeline.py tab is currently active, displaying the following Python code:

```
18     def extract_structured_sections(filepath, toc_lookup=None): 1 usage
83         chunks.append({
84             "text": para_group,
85             "page": page_num,
86             "source": os.path.basename(filepath),
87             "section": current_section,
88             "type": "paragraph-group",
89             "images": images,
90             "equations": eqs,
91             "mode": "page-paragraph"
92         })
93
94     doc.close()
95     return chunks
96
97
98     def stitch_by_section(texts, metadata): 1 usage
99         from collections import defaultdict
100        grouped = defaultdict(list)
101
102        for text, meta in zip(texts, metadata):
103            key = meta.get("section", "Unknown")
104            grouped[key].append((text, meta))
105
106        stitched_texts = []
107        stitched_metadata = []
108
109        for section, chunks in grouped.items():
110            combined_text = "\n\n".join([chunk[0] for chunk in chunks])
111            base_meta = chunks[0][1].copy()
112            base_meta["stitched_chunks"] = len(chunks)
113            base_meta["type"] = "stitched"
114            stitched_texts.append(combined_text)
115            stitched_metadata.append(base_meta)
116
117    return stitched_texts, stitched_metadata
```

```

98     def stitch_by_section(texts, metadata): 1 usage
116
117         return stitched_texts, stitched_metadata
118
119
120     def build_index(): 4 usages
121         if os.path.exists(VECTOR_DB_PATH):
122             os.remove(VECTOR_DB_PATH)
123             print("☒ Old vector index deleted.")
124
125         all_chunks, metadata = [], []
126         toc_titles = None
127
128         for i, file in enumerate(sorted(os.listdir("data/raw"))):
129             if not file.endswith(".pdf"):
130                 continue
131
132             path = os.path.join("data/raw", file)
133
134             if toc_titles is None and "LectureNotes-1.pdf" in file:
135                 from ingest.toc_parser import parse_toc_file
136                 toc_titles = parse_toc_file("./data/CompMechanicsLectureNotesTOC.toc")
137                 with open(TOC_CACHE_PATH, "wb") as f:
138                     pickle.dump(toc_titles, f)
139                     print(f"■ TOC parsed with {len(toc_titles)} entries.")
140             elif toc_titles is None:
141                 with open(TOC_CACHE_PATH, "rb") as f:
142                     toc_titles = pickle.load(f)
143                     print(f"■ TOC loaded from cache with {len(toc_titles)} entries.")
144
145             try:
146                 sections = extract_structured_sections(path, toc_titles)
147             except Exception as e:
148                 print(f"⚠ Error processing {file}: {e}")
149                 continue

```

```

120     def build_index(): 4 usages
121         print(f"⚠ Error processing {file}: {e}")
122         continue
123
124         for i, section in enumerate(sections):
125             section_number = next((entry["number"] for entry in toc_titles if entry["title"] == section["section"]), "Unknown")
126             metadata.append({
127                 "chunk_id": f"{file}-{p}{section['page']}-c{i + 1}",
128                 "page": section["page"],
129                 "source": section["source"],
130                 "section": section["section"],
131                 "section_number": section_number,
132                 "type": section["type"],
133                 "images": section["images"],
134                 "equations": section["equations"],
135                 "mode": section["mode"]
136             })
137         all_chunks.append(section["text"])
138
139         if not all_chunks:
140             print("⚠ No valid chunks extracted. Index not created.")
141             return
142
143         all_chunks, metadata = stitch_by_section(all_chunks, metadata)
144         vectors = embed_chunks(all_chunks)
145
146         vs = VectorStore(dim=vectors.shape[1])
147         vs.add(vectors, all_chunks, metadata)
148
149         os.makedirs(os.path.dirname(VECTOR_DB_PATH), exist_ok=True)
150         with open(VECTOR_DB_PATH, "wb") as f:
151             pickle.dump(vs, f)
152
153         print(f"☒ Index built with {len(all_chunks)} stitched sections.")

```

Afterwards I worked on the methodology for a little bit. I still have much to do, however.

## Thursday July 24th 7/24/25

Goals: Work more on implementing some best practices improvements to the bot. Work more on methodology (see what I can get done), probably start with extra improvements.

I spent most of the morning implementing section boosting (boosting sections with titles/keywords related to sections specifically mentioned within the query). I compared relevant chunk answers from before boosting, after boosting, and after adding a soft-filtering/postreranking system. There was not much difference between any of them, however ChatGPT says the new system would help with broader, more vague questions. It said I already have strong vector embeddings that work very well, so this did not have a strong impact on that.

Later today I might continue working and seeing what else I could implement. I would like to finish my methodology today (or at least do a lot of it), so we'll see how much time I have.

From clarifying with Andrew, I'm going to continue working on project enhancements. Probably around 4:00 I'll work more on the rough draft.

After talking w/ Albert decided on a new direction.

I reverted the recent changes made today, and I'm at the point after the TOC parser but before the section boosting. Essentially the end of yesterday. I'm going to try and implement the rest of the lecture notes to see how, or if, it works.

I also changed the number of chunks searched and pulled. It searches the 30 *top\_k* chunks and pulls 5 related to the query vector.

## Friday July 25th 7/25/25

I changed the filtering of emails and academic affiliations. I realized that before, entire blocks of text featuring these were being skipped instead of only the lines necessary. I changed this to fit only the lines. I also added a global page offset, so output chunks show the page relative to the entire lecture notes, not the page of the split PDF.

Preprocess\_pipeline.py build\_index() function:

```
retriever.py    rag_project.py    embedder.py    extract_sent_chunks.py    toc_parser.py    app.py    preprocess_pipeline.py

137     def build_index():
138         if os.path.exists(VECTOR_DB_PATH):
139             os.remove(VECTOR_DB_PATH)
140             print("Old vector index deleted.")
141
142         all_chunks, metadata = [], []
143         toc_titles = None
144
145         # ✅ Mapping for page offset by file
146         start_page_offset = {
147             "Jeremic_et_al_CompMech_LectureNotes-1.pdf": 1,
148             "Jeremic_et_al_CompMech_LectureNotes-2.pdf": 301,
149             "Jeremic_et_al_CompMech_LectureNotes-3.pdf": 601,
150             "Jeremic_et_al_CompMech_LectureNotes-4.pdf": 901,
151             "Jeremic_et_al_CompMech_LectureNotes-5.pdf": 1201,
152             "Jeremic_et_al_CompMech_LectureNotes-6.pdf": 1501,
153             "Jeremic_et_al_CompMech_LectureNotes-7.pdf": 1801,
154             "Jeremic_et_al_CompMech_LectureNotes-8.pdf": 2101,
155             "Jeremic_et_al_CompMech_LectureNotes-9.pdf": 2401,
156             "Jeremic_et_al_CompMech_LectureNotes-10.pdf": 2701,
157             "Jeremic_et_al_CompMech_LectureNotes-11.pdf": 3001
158         }
159
160         for i, file in enumerate(sorted(os.listdir("data/raw"))):
161             if not file.endswith(".pdf"):
162                 continue
163
164             path = os.path.join("data/raw", file)
165
166             if toc_titles is None and "LectureNotes-1.pdf" in file:
167                 from ingest.toc_parser import parse_toc_file
168                 toc_titles = parse_toc_file("./data/CompMechanicsLectureNotesTOC.toc")
169                 with open(TOC_CACHE_PATH, "wb") as f:
170                     pickle.dump(toc_titles, f)
171                 print(f"toc parsed with {len(toc_titles)} entries.")
172             elif toc_titles is None:
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
```

```
retriever.py    rag_project.py    embedder.py    extract_sent_chunks.py    toc_parser.py    app.py    preprocess_pipeline.py ×

137     def build_index():
138         if toc_titles is None and "LectureNotes-1.pdf" in file:
139             from ingest.toc_parser import parse_toc_file
140             toc_titles = parse_toc_file("./data/CompMechanicsLectureNotesTOC.toc")
141             with open(TOC_CACHE_PATH, "wb") as f:
142                 pickle.dump(toc_titles, f)
143             print(f"toc parsed with {len(toc_titles)} entries.")
144         elif toc_titles is None:
145             with open(TOC_CACHE_PATH, "rb") as f:
146                 toc_titles = pickle.load(f)
147             print(f"toc loaded from cache with {len(toc_titles)} entries.")
148
149         # ✅ Filter TOC entries to match current file's local page range
150         doc = fitz.open(path)
151         page_count = doc.page_count
152         doc.close()
153
154         filtered_toc = [entry for entry in toc_titles if 0 <= entry["page"] < page_count]
155
156         try:
157             sections = extract_structured_sections(path, filtered_toc)
158         except Exception as e:
159             print(f"⚠ Error processing {file}: {e}")
160             continue
161
162         offset = start_page_offset.get(file, 0)
163
164         for i, section in enumerate(sections):
165             local_page = section.get("page", 0)
166             global_page = offset + local_page
167
168             section_number = next((entry["number"] for entry in toc_titles if entry["title"] == section["section"]), "Unknown")
169             metadata.append({
170                 "chunk_id": f"{file}-p{section['page']}-c{i + 1}",
171                 "page": global_page,
172                 "source": section["source"],
173             })
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
```

```
retriever.py    rag_project.py    embedder.py    extract_sent_chunks.py    toc_parser.py    app.py    preprocess_pipeline.py x
.37     def build_index():
.38         local_page = section.get("page", 0)
.39         global_page = offset + local_page
.40
.41         section_number = next((entry["number"] for entry in toc_titles if entry["title"] == section["section"]), "Unknown")
.42         metadata.append({
.43             "chunk_id": f"{file}-{p}{section['page']}-{c}{i + 1}",
.44             "page": global_page,
.45             "source": section["source"],
.46             "section": section["section"],
.47             "section_number": section_number,
.48             "type": section["type"],
.49             "images": section["images"],
.50             "equations": section["equations"],
.51             "mode": section["mode"]
.52         })
.53         all_chunks.append(section["text"])
.54
.55     if not all_chunks:
.56         print("⚠ No valid chunks extracted. Index not created.")
.57         return
.58
.59     all_chunks, metadata = stitch_by_section(all_chunks, metadata)
.60     vectors = embed_chunks(all_chunks)
.61
.62     vs = VectorStore(dim=vectors.shape[1])
.63     vs.add(vectors, all_chunks, metadata)
.64
.65     os.makedirs(os.path.dirname(VECTOR_DB_PATH), exist_ok=True)
.66     with open(VECTOR_DB_PATH, "wb") as f:
.67         pickle.dump(vs, f)
.68
.69     print(f"✅ Index built with {len(all_chunks)} stitched sections.")
```

Preprocess\_pipeline.py extract\_structured\_sections() function:

```
retriever.py    rag_project.py    embedder.py    extract_sent_chunks.py    toc_parser.py    app.py    preprocess_pipeline.py ×
26
27     def extract_structured_sections(filepath, toc_lookup=None):  1 usage
28         doc = fitz.open(filepath)
29         chunks = []
30         seen_texts = set()
31         current_section = "Unknown Section"
32
33         section_by_page = {}
34         if toc_lookup:
35             sorted_toc = sorted(toc_lookup, key=lambda x: x["page"])
36             for i, entry in enumerate(sorted_toc):
37                 start = entry["page"]
38                 end = sorted_toc[i + 1]["page"] if i + 1 < len(sorted_toc) else 10_000
39                 for p in range(start, end):
40                     section_by_page[p] = entry["title"]
41
42         for page_num, page in enumerate(doc, start=1):
43             current_section = section_by_page.get(page_num, current_section)
44
45             blocks = page.get_text("dict")["blocks"]
46             images = [{"ext": img[1], "xref": img[0]} for img in page.get_images(full=True)]
47
48             page_paragraphs = []
49             for block in blocks:
50                 if "lines" not in block:
51                     continue
52
53                 cleaned_lines = []
54                 for line in block["lines"]:
55                     line_text = " ".join(span["text"] for span in line["spans"]).strip()
56                     if not line_text or len(line_text) < 10:
57                         continue
58                     if EMAIL_PATTERN.search(line_text.lower()) or AFFILIATION_PATTERN.search(line_text.lower()):
59                         continue
60                     cleaned_lines.append(line_text)
```

```
retriever.py    rag_project.py    embedder.py    extract_sent_chunks.py    toc_parser.py    app.py    preprocess_pipeline.py ×
27     def extract_structured_sections(filepath, toc_lookup=None):  1 usage
28         text = " ".join(cleaned_lines).strip()
29         if not text or len(text) < 10:
30             continue
31
32         if SECTION_HEADER_PATTERN.match(text):
33             current_section = text.strip()
34             continue
35
36         page_paragraphs.append(text)
37
38         # Paragraph grouping
39         grouped = []
40         current_group = []
41         word_count = 0
42
43         for para in page_paragraphs:
44             wc = len(para.split())
45             if para in seen_texts or wc < 10:
46                 continue
47             seen_texts.add(para)
48
49             current_group.append(para)
50             word_count += wc
51
52             if word_count >= 300:
53                 grouped.append(" ".join(current_group))
54                 current_group = []
55                 word_count = 0
56
57             if current_group:
58                 grouped.append(" ".join(current_group))
59
60         for para_group in grouped:
61             eqs = []
62             if "=" in para_group or any(w in para_group for w in ["\n", "\t", "\r", "\f", "\u0333", "\u0331", "\u0332", "\u0334", "\u0335", "\u0336", "\u0337", "\u0338", "\u0339", "\u033a", "\u033b", "\u033c", "\u033d", "\u033e", "\u033f", "\u033g", "\u033h", "\u033i", "\u033j", "\u033k", "\u033l", "\u033m", "\u033n", "\u033o", "\u033p", "\u033q", "\u033r", "\u033s", "\u033t", "\u033u", "\u033v", "\u033w", "\u033x", "\u033y", "\u033z", "\u033A", "\u033B", "\u033C", "\u033D", "\u033E", "\u033F", "\u033G", "\u033H", "\u033I", "\u033J", "\u033K", "\u033L", "\u033M", "\u033N", "\u033O", "\u033P", "\u033Q", "\u033R", "\u033S", "\u033T", "\u033U", "\u033V", "\u033W", "\u033X", "\u033Y", "\u033Z"]):
```

A screenshot of a code editor interface. At the top, there is a horizontal bar with several tabs, each represented by a small icon and the name of a Python file: retriever.py, rag\_project.py, embedder.py, extract\_sent\_chunks.py, toc\_parser.py, app.py, and preprocess\_pipeline.py. The tab for 'app.py' is currently active, indicated by a blue underline. Below the tabs, the main workspace displays a block of Python code. The code is a function named 'extract\_structured\_sections' located in the 'app.py' file. The code uses a for loop to iterate through 'page\_paragraphs'. It tracks word counts and groups paragraphs into 'current\_group'. When the word count reaches 300 or if an equation symbol like '=' is found in the group, it appends the current group to 'grouped' and initializes 'current\_group' and 'word\_count' again. Finally, it appends the last 'current\_group' to 'grouped'. A list 'eqs' is used to store groups containing equations. The code then iterates through 'grouped' to create 'chunks'. Each chunk is a dictionary with keys: 'text' (the paragraph group), 'page' (page number), 'source' (file path), 'section' (current section), 'type' ('paragraph-group'), 'images' (not used in this snippet), 'equations' (the 'eqs' list), and 'mode' ('page-paragraph'). The code concludes with 'doc.close()' and a return statement for 'chunks'.

```
27     def extract_structured_sections(filepath, toc_lookup=None): 1usage
77         for para in page_paragraphs:
78             wc = len(para.split())
79             if para in seen_texts or wc < 10:
80                 continue
81             seen_texts.add(para)
82
83             current_group.append(para)
84             word_count += wc
85
86             if word_count >= 300:
87                 grouped.append(" ".join(current_group))
88                 current_group = []
89                 word_count = 0
90
91             if current_group:
92                 grouped.append(" ".join(current_group))
93
94         for para_group in grouped:
95             eqs = []
96             if "=" in para_group or any(w in para_group for w in ["\u2248", "\u0333", "\u0335", "\u0337", "\u0339", "\u033a", "\u033b"]):
97                 eqs.append(para_group)
98
99             chunks.append({
100                 "text": para_group,
101                 "page": page_num,
102                 "source": os.path.basename(filepath),
103                 "section": current_section,
104                 "type": "paragraph-group",
105                 "images": images,
106                 "equations": eqs,
107                 "mode": "page-paragraph"
108             })
109
110         doc.close()
111     return chunks
```