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)

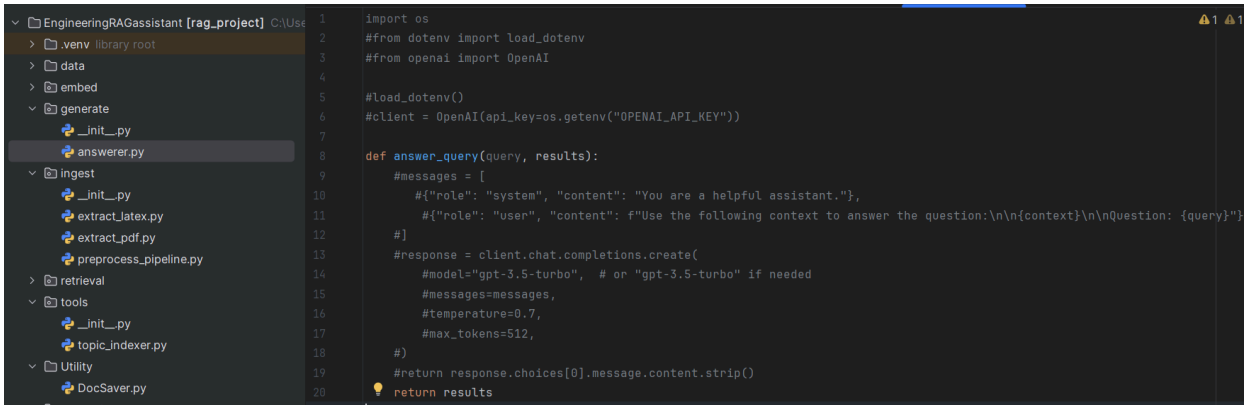
Rag file structure at end of day 2 of coding:

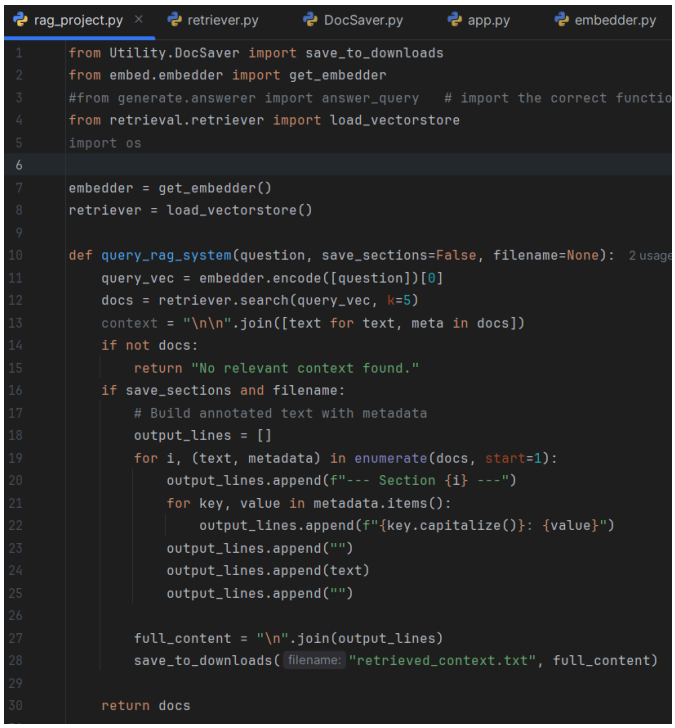
```
EngineeringRAGassistant/
├── app.py
├── rag_project.py
├── data/
│   └── raw/
├── vector_index/
│   └── index.pkl
├── ingest/
│   ├── __init__.py
│   ├── extract_pdf.py
│   ├── extract_latex.py
│   └── preprocess_pipeline.py
├── embed/
│   └── embedder.py
```

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

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

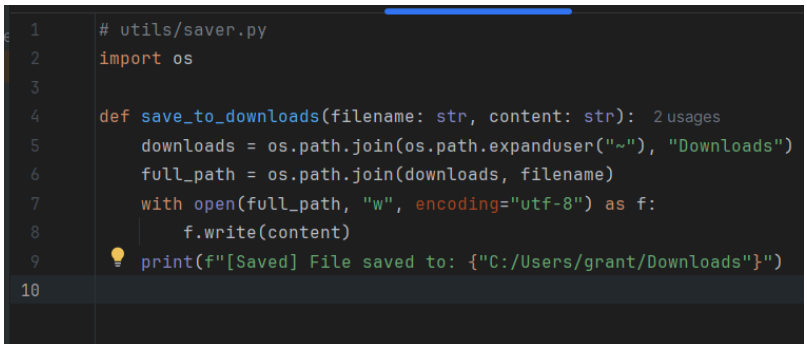
Grant: Tried using ChatGPT from inside bot. Received API key from openAI, however I ran out of access to GPT 4o and tried using GPT 3.5 turbo instead. Ran into the problem of having to pay for more tokens in order to have the retriever and generator condensed into a single script. Decided to shift into 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.





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.

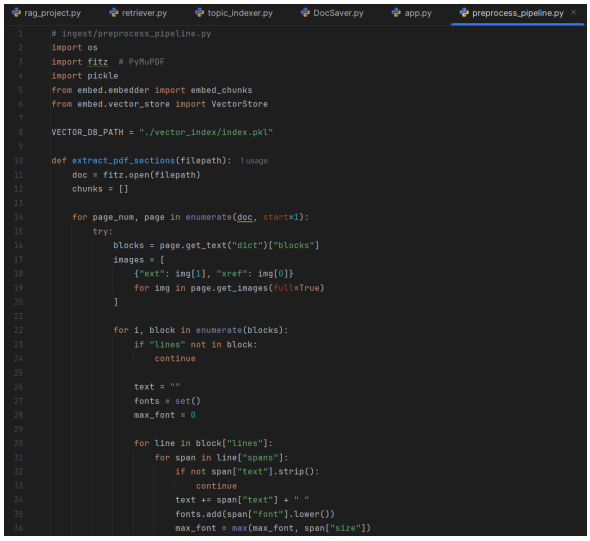


```python
# utils/saver.py
import os

def save_to_downloads(filename: str, content: str):  2 usages
    downloads = os.path.join(os.path.expanduser("~"), "Downloads")
    full_path = os.path.join(downloads, filename)
    with open(full_path, "w", encoding="utf-8") as f:
        f.write(content)
    print(f"[Saved] File saved to: {"C:/Users/grant/Downloads"}")

```

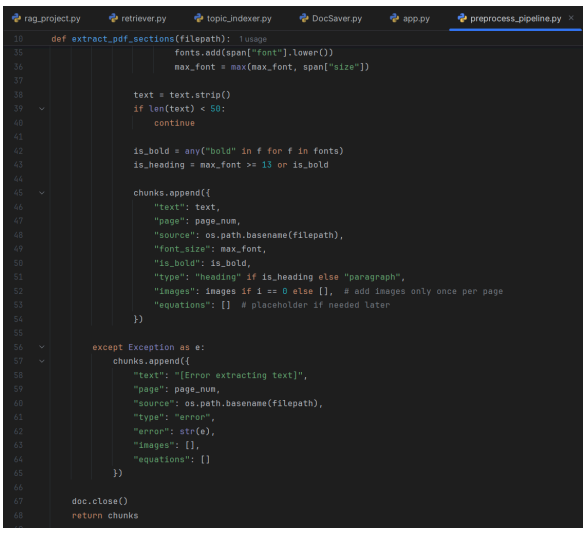
<u>Tuesday, July 15th (7/15/25)</u>
Grant:
Instructions as per Dr. Jeremic: Test DeepSeek, test inputting random/strange questions or characters, work on indexing, and validate output responses from AI
Personal Goals: Mess with topic indexer

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

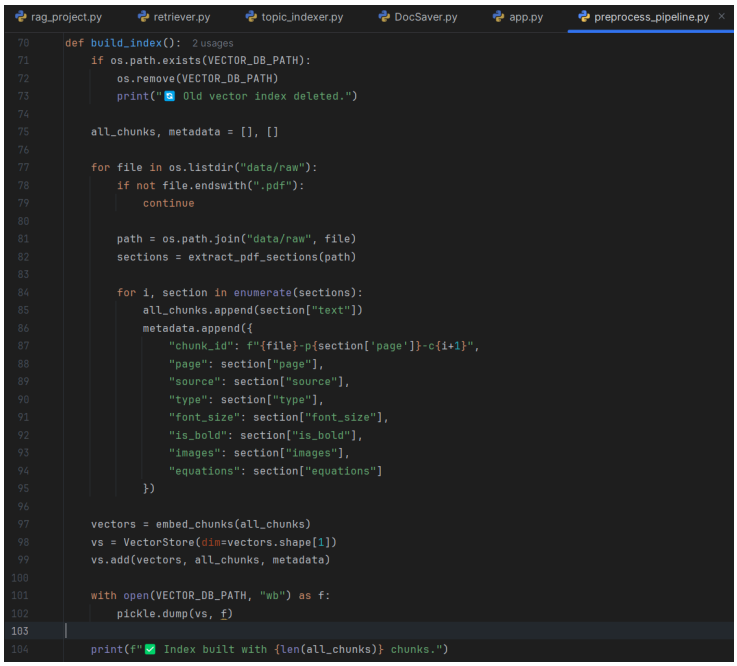
The chunks pulled from the pdf are the same number as the number of pages, so I will try to make more, smaller chunks related to the sections of the paper.
Using the already imported fitz or PyMuPDF extractor tool, I made the chunks per section rather than per page. I moved the PDF extraction from extract_pdf.py all to the preprocess_pipeline and I still included the image, text, and equation metadata. I will test the functionality of this further.



```python
# ingest/preprocess_pipeline.py
import os
import fitz  # PyMuPDF
import pickle
from embed.embedder import embed_chunks
from embed.vector_store import VectorStore

VECTOR_DB_PATH = "./vector_index/index.pkl"

def extract_pdf_sections(filepath):  1 usage
    doc = fitz.open(filepath)
    chunks = []

    for page_num, page in enumerate(doc, start=1):
        try:
            blocks = page.get_text("dict")["blocks"]
            images = [
                {"ext": img[1], "xref": img[0]}
                for img in page.get_images(full=True)
            ]

            for i, block in enumerate(blocks):
                if "lines" not in block:
                    continue

                text = ""
                fonts = set()
                max_font = 0

                for line in block["lines"]:
                    for span in line["spans"]:
                        if not span["text"].strip():
                            continue
                        text += span["text"] + " "
                        fonts.add(span["font"].lower())
                        max_font = max(max_font, span["size"])
```



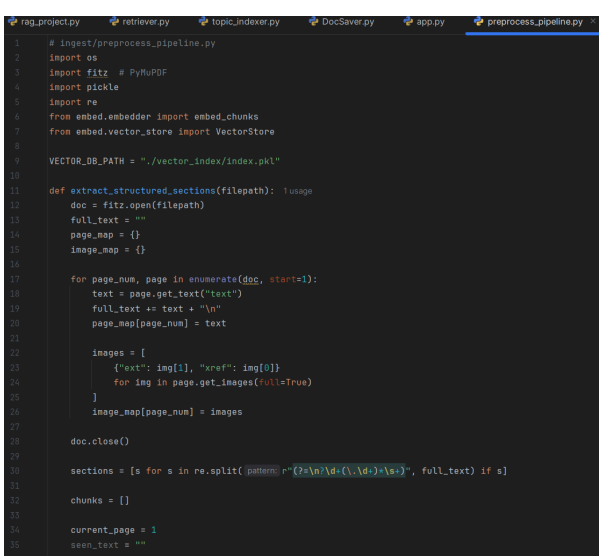
```python
def extract_pdf_sections(filepath):  1 usage
                        fonts.add(span["font"].lower())
                        max_font = max(max_font, span["size"])

                text = text.strip()
                if len(text) < 50:
                    continue

                is_bold = any("bold" in f for f in fonts)
                is_heading = max_font >= 13 or is_bold

                chunks.append({
                    "text": text,
                    "page": page_num,
                    "source": os.path.basename(filepath),
                    "font_size": max_font,
                    "is_bold": is_bold,
                    "type": "heading" if is_heading else "paragraph",
                    "images": images if i == 0 else [],  # add images only once per page
                    "equations": []  # placeholder if needed later
                })

        except Exception as e:
            chunks.append({
                "text": "[Error extracting text]",
                "page": page_num,
                "source": os.path.basename(filepath),
                "type": "error",
                "error": str(e),
                "images": [],
                "equations": []
            })

    doc.close()
    return chunks
```



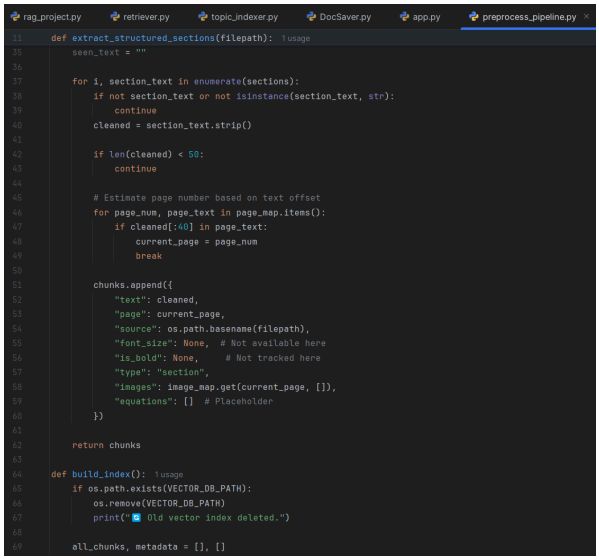
```python
def build_index():  2 usages
    if os.path.exists(VECTOR_DB_PATH):
        os.remove(VECTOR_DB_PATH)
        print("✅ Old vector index deleted.")

    all_chunks, metadata = [], []

    for file in os.listdir("data/raw"):
        if not file.endswith(".pdf"):
            continue

        path = os.path.join("data/raw", file)
        sections = extract_pdf_sections(path)

        for i, section in enumerate(sections):
            all_chunks.append(section["text"])
            metadata.append({
                "chunk_id": f"{file}-p{section['page']}-c{i+1}",
                "page": section["page"],
                "source": section["source"],
                "type": section["type"],
                "font_size": section["font_size"],
                "is_bold": section["is_bold"],
                "images": section["images"],
                "equations": section["equations"]
            })

    vectors = embed_chunks(all_chunks)
    vs = VectorStore(dim=vectors.shape[1])
    vs.add(vectors, all_chunks, metadata)

    with open(VECTOR_DB_PATH, "wb") as f:
        pickle.dump(vs, f)

    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.

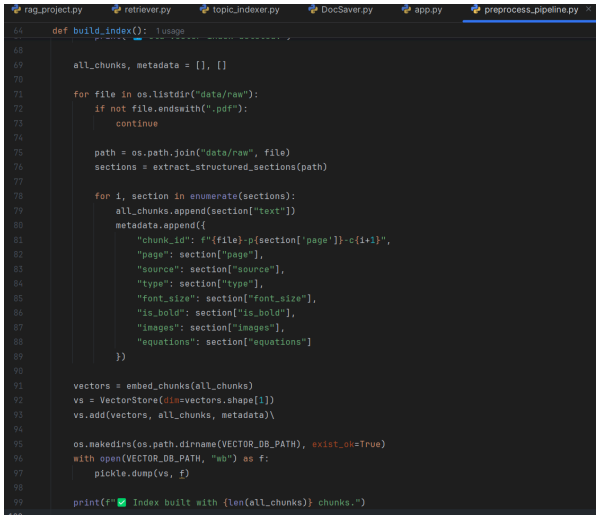
I updated the code for semantic chunking per labeled section (as there are in scientific papers), however an attribute error occurred. I fixed the attribute error by skipping stray line breaks or malformed matches. I bugfixed some file errors, but still was retrieving irrelevant chunks from the PDF.



```python
# ingest/preprocess_pipeline.py
import os
import fitz  # PyMuPDF
import pickle
import re
from embed.embedder import embed_chunks
from embed.vector_store import VectorStore

VECTOR_DB_PATH = "./vector_index/index.pkl"

def extract_structured_sections(filepath):  1 usage
    doc = fitz.open(filepath)
    full_text = ""
    page_map = {}
    image_map = {}

    for page_num, page in enumerate(doc, start=1):
        text = page.get_text("text")
        full_text += text + "\n"
        page_map[page_num] = text

        images = [
            {"ext": img[1], "xref": img[0]}
            for img in page.get_images(full=True)
        ]
        image_map[page_num] = images

    doc.close()

    sections = [s for s in re.split( pattern: r"(?=\n?\d+(\.\d+)+\s+)", full_text) if s]

    chunks = []

    current_page = 1
    seen_text = ""
```



```python
def extract_structured_sections(filepath):  1 usage
    seen_text = ""

    for i, section_text in enumerate(sections):
        if not section_text or not isinstance(section_text, str):
            continue
        cleaned = section_text.strip()

        if len(cleaned) < 50:
            continue

        # Estimate page number based on text offset
        for page_num, page_text in page_map.items():
            if cleaned[:40] in page_text:
                current_page = page_num
                break

        chunks.append({
            "text": cleaned,
            "page": current_page,
            "source": os.path.basename(filepath),
            "font_size": None,  # Not available here
            "is_bold": None,    # Not tracked here
            "type": "section",
            "images": image_map.get(current_page, []),
            "equations": []  # Placeholder
        })

    return chunks

def build_index():  1 usage
    if os.path.exists(VECTOR_DB_PATH):
        os.remove(VECTOR_DB_PATH)
        print("🗑️ Old vector index deleted.")

    all_chunks, metadata = [], []
```

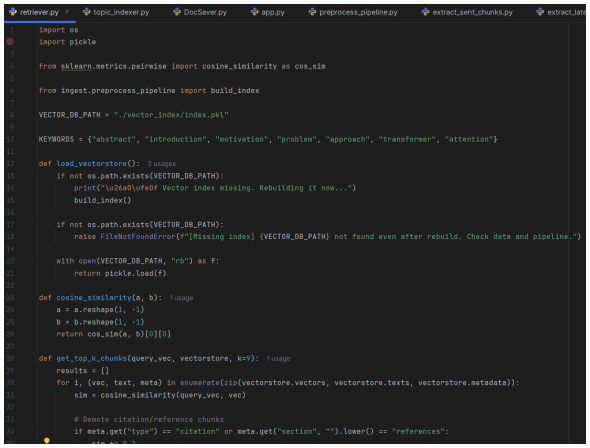


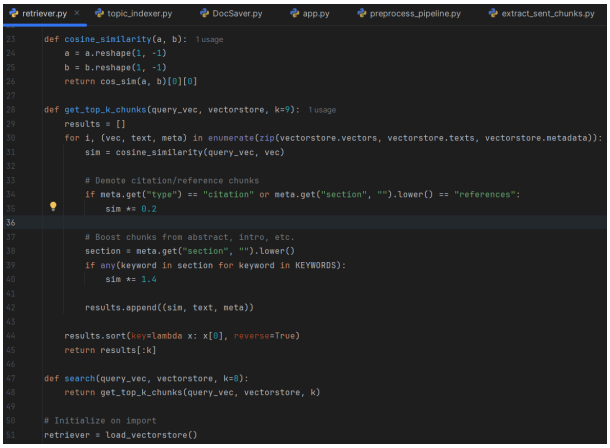
```python
def build_index():  1 usage

    all_chunks, metadata = [], []

    for file in os.listdir("data/raw"):
        if not file.endswith(".pdf"):
            continue

        path = os.path.join("data/raw", file)
        sections = extract_structured_sections(path)

        for i, section in enumerate(sections):
            all_chunks.append(section["text"])
            metadata.append({
                "chunk_id": f"{file}-p{section['page']}-c{i+1}",
                "page": section["page"],
                "source": section["source"],
                "type": section["type"],
                "font_size": section["font_size"],
                "is_bold": section["is_bold"],
                "images": section["images"],
                "equations": section["equations"]
            })

    vectors = embed_chunks(all_chunks)
    vs = VectorStore(dim=vectors.shape[1])
    vs.add(vectors, all_chunks, metadata)\

    os.makedirs(os.path.dirname(VECTOR_DB_PATH), exist_ok=True)
    with open(VECTOR_DB_PATH, "wb") as f:
        pickle.dump(vs, f)

    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.

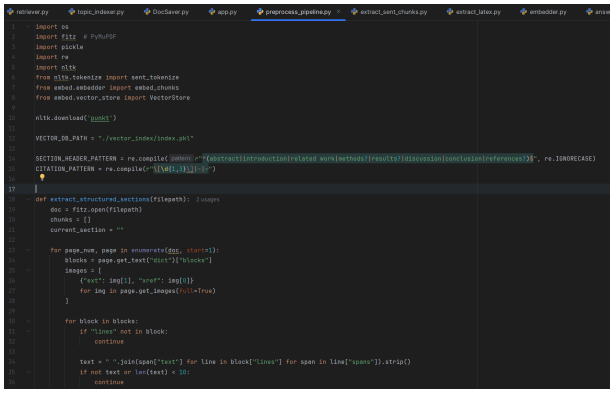
I moved all of preprocess_pipeline to sentence-based chunking with window overlap, citation filtering, and tagged section headers. I ran into a bug with the NLTK download, but it was fixed. I added more citation and reference filtering to the bot as well as some abstract keyword boosting. I remade retriever.py 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)

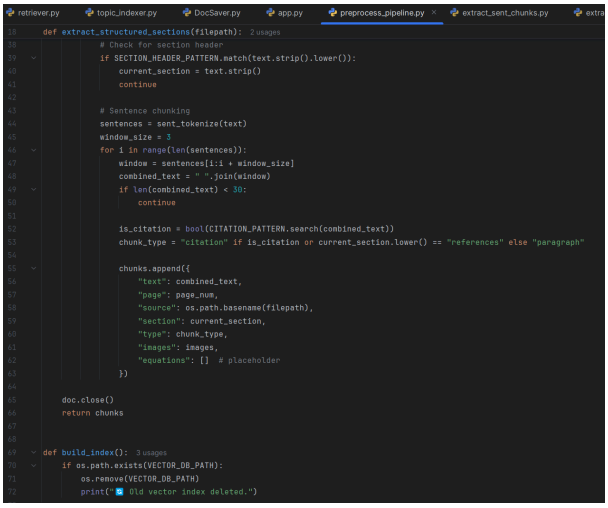
Retriever.py:





Preprocess_pipeline.py:



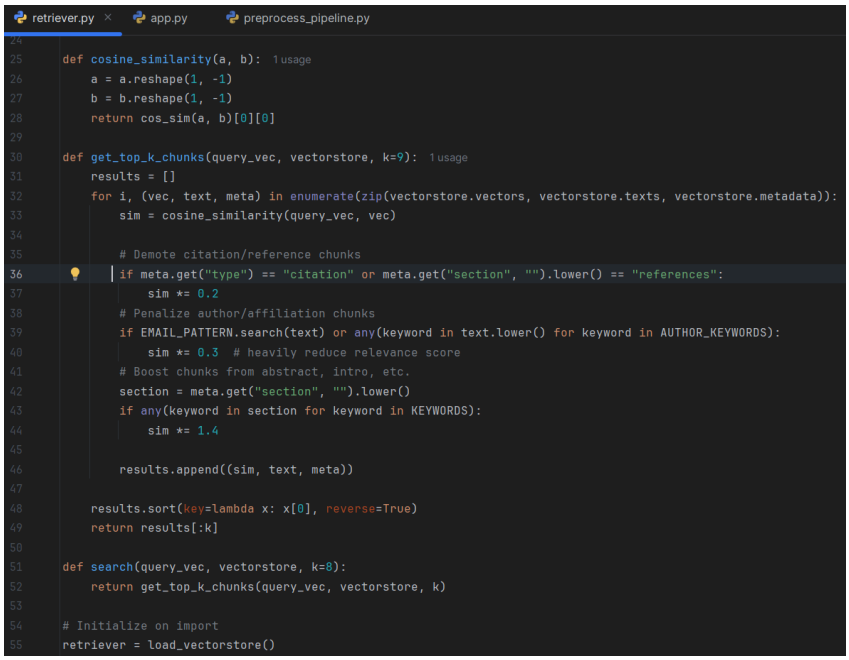


The number of chunks was moved up to 9 due to how small the chunks are now. The reference/citation filtering/nullifying seems to have helped a lot, but semantic relevance can still be improved. The relevance has definitely improved since the previous page-based chunking.

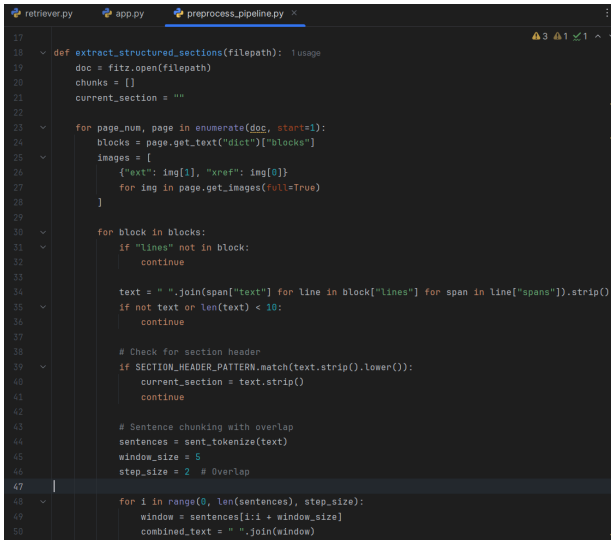
Wednesday July 16 (7/16/25)

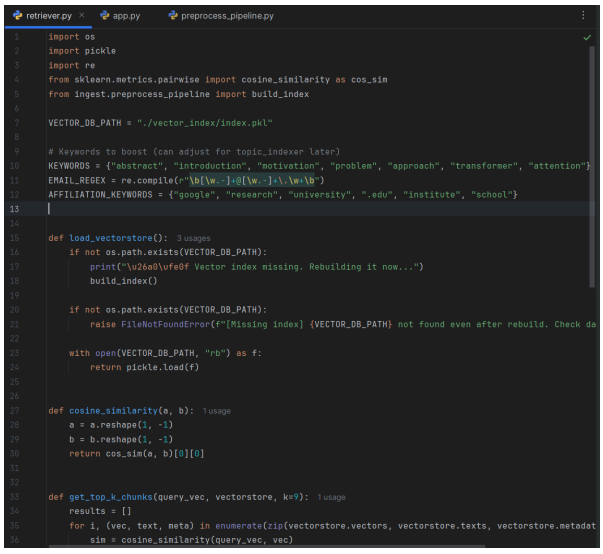
Goals: Mess with keyword boosting and citation filtering. Try using topic_indexer potentially. Start integrating parts of the lecture notes to specialize the bot towards that style of literature, not strictly this scientific paper.

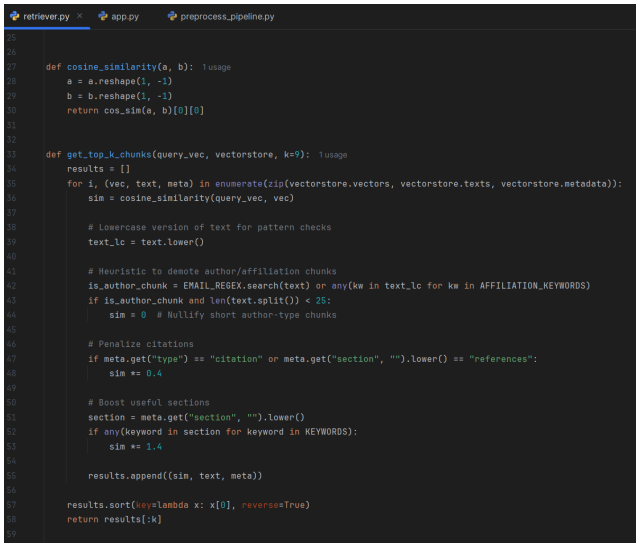
I started by adding some author and email keywords and adding a relevance penalizer to author names and emails.



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.

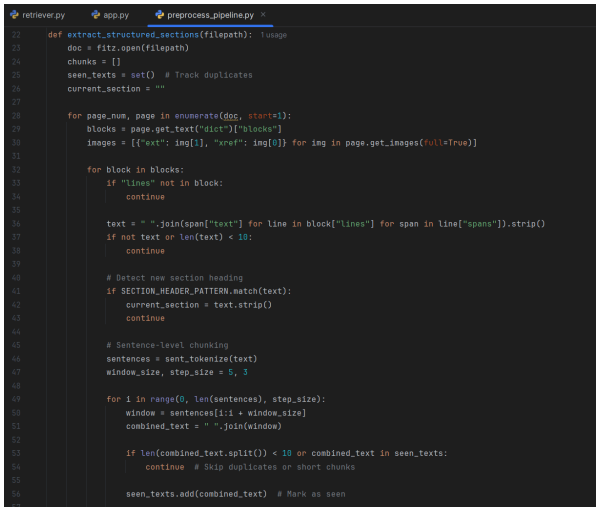


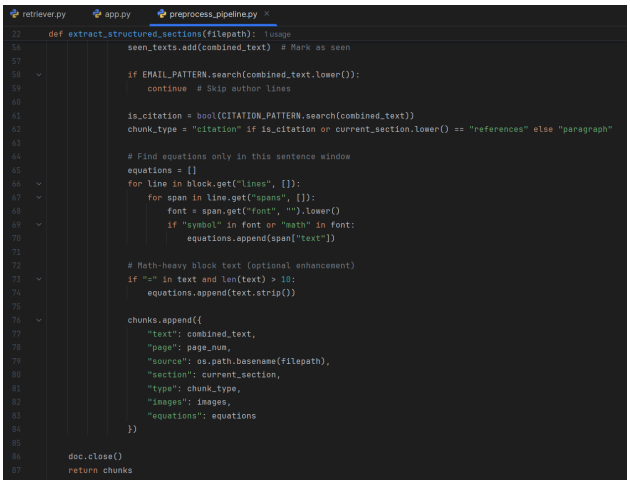




Before integrating the first section of the lecture notes, I asked ChatGPT for recommendations on the chunking approach. It course corrected to a hybrid, section-sentence based chunking

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.





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%|[redacted]| 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. Convolu- tional 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

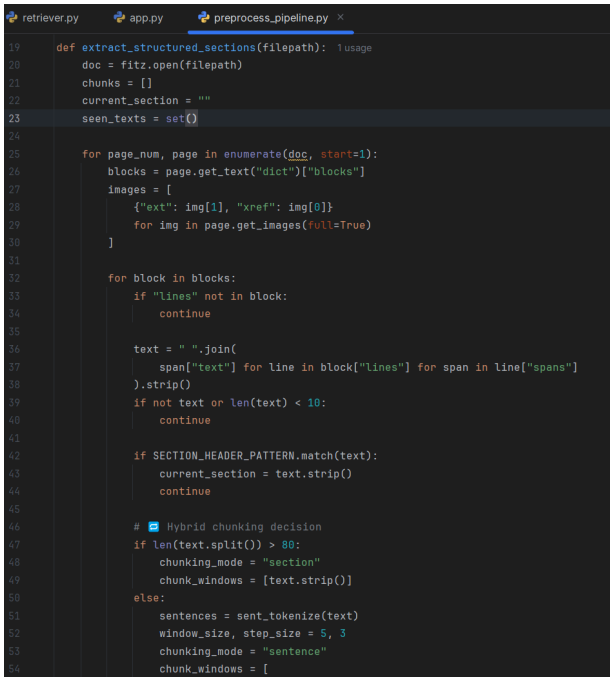
We also experimented with using learned positional embeddings [9] instead, and found that the two versions produced nearly identical results (see Table 3 row (E)). We chose the sinusoidal version because it may allow the model to extrapolate to sequence lengths longer than the ones encountered during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

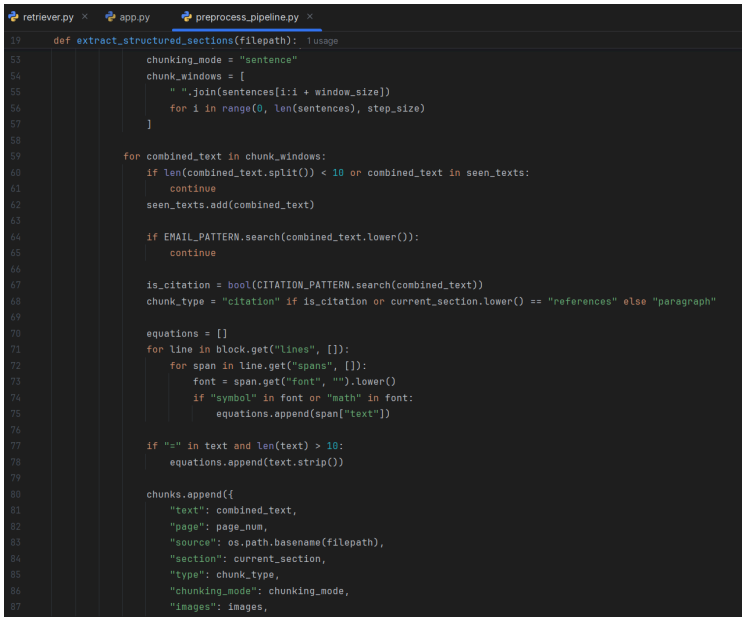
---

[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





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.

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

```python
# toc_parser.py

import fitz  # PyMuPDF
import re

TOC_SECTION_PATTERN = re.compile( pattern: r"^((chapter|section|part)?\s*\d+(\.\d+)*)(\s+.+)?$", re.IGNORECASE)

def parse_toc(filepath, max_pages=89):  2 usages
    doc = fitz.open(filepath)
    toc_sections = []

    for page_num in range(min(max_pages, len(doc))):
        page = doc[page_num]
        blocks = page.get_text("dict")["blocks"]

        for block in blocks:
            if "lines" not in block:
                continue
            text = " ".join(span["text"] for line in block["lines"] for span in line["spans"]).strip()
            if TOC_SECTION_PATTERN.match(text):
                toc_sections.append(text)

    doc.close()
    return toc_sections

```

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.

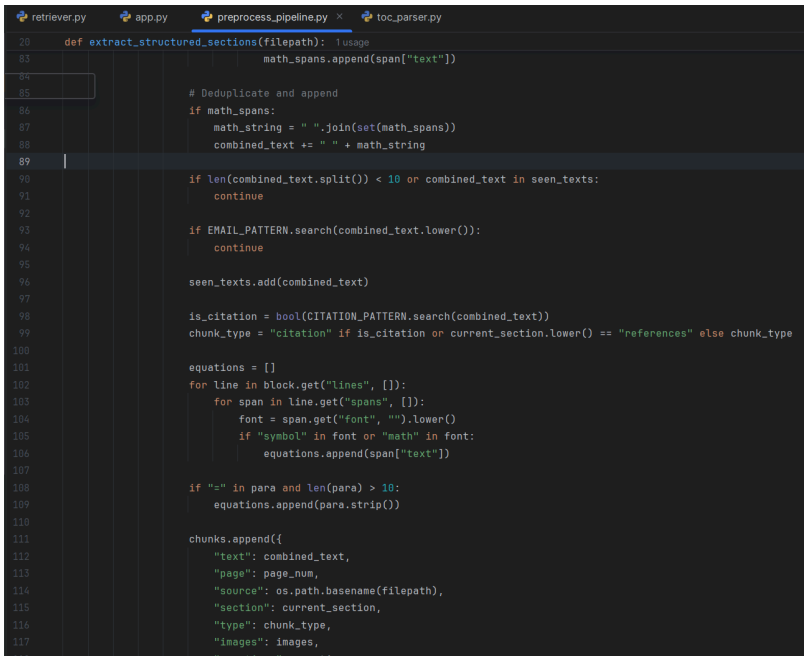
Before starting, I asked ChatGPT for any final recommendations of what to add. It recommended using style header recognition, paragraph merging, and figure/table caption preservation. I added those all to the code. I also asked ChatGPT about page-chunking again due to how long each of the subsections 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

```python
EMAIL_PATTERN = re.compile(r"@[\.edu|gmail\.com|google\.com|research")
FIGURE_PATTERN = re.compile( pattern: r"(figure|table)\s+\d+(\.\d+)*", re.IGNORECASE)


def extract_structured_sections(filepath):  1 usage
    doc = fitz.open(filepath)
    chunks = []
    current_section = ""
    seen_texts = set()

    for page_num, page in enumerate(doc, start=1):
        blocks = page.get_text("dict")["blocks"]
        images = [
            {"ext": img[1], "xref": img[0]}
            for img in page.get_images(full=True)
        ]

        for block in blocks:
            if "lines" not in block:
                continue

            lines = [" ".join(span["text"] for span in line["spans"]).strip() for line in block["lines"]]
            paragraphs = []
            current_para = ""

            # Merge broken lines into paragraphs
            for line in lines:
                if not line:
                    continue
                if line[-1] in ".:;?":
                    current_para += " " + line
                    paragraphs.append(current_para.strip())
                    current_para = ""
                else:
                    current_para += " " + line
            if current_para:
```

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

```python
    def extract_structured_sections(filepath):  1 usage
                    current_para += " " + line
            if current_para:
                paragraphs.append(current_para.strip())

            for para in paragraphs:
                if not para or len(para) < 10:
                    continue

                # Detect new section heading
                if SECTION_HEADER_PATTERN.match(para):
                    current_section = para.strip()
                    continue

                # Detect and store figure/table captions
                if FIGURE_PATTERN.match(para):
                    chunk_type = "caption"
                else:
                    chunk_type = "paragraph"

                # Sentence chunking
                sentences = sent_tokenize(para)
                window_size, step_size = 5, 3
                for i in range(0, len(sentences), step_size):
                    window = sentences[i:i + window_size]
                    combined_text = " ".join(window)

                    # Find math spans in the block and append them to the chunk if they weren't already included
                    math_spans = []
                    for line in block.get("lines", []):
                        for span in line.get("spans", []):
                            font = span.get("font", "").lower()
                            if "symbol" in font or "math" in font or re.search( pattern: r"[\d\w]+\s*=\s*[\d\w]",
                                                                               span["text"]):  # crude math check
                                math_spans.append(span["text"])
```



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.

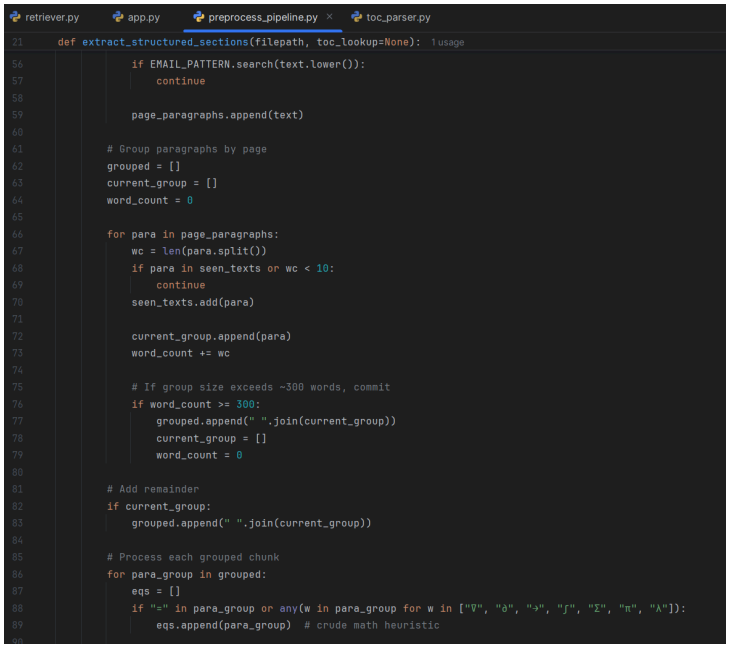
(I also ran into issues with unknown conversation limitations. After such a long conversation on the plus account I was not able to ask any more queries. I googled it and it is supposedly due to high traffic times during the day and a maximum amount of queries within 3 hours. Needless to say I made a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

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

```python
import fitz
import re

TOC_ROW_PATTERN = re.compile(r"^(\d+(\.\d+)+)\s+(.+?)\s+(\d+)$")

def parse_toc(filepath, max_pages=89):  2 usages
    doc = fitz.open(filepath)
    toc = []
    candidate_lines = []

    for page in doc[:max_pages]:
        lines = page.get_text("text").split("\n")
        for line in lines:
            candidate_lines.append(line.strip())

    # Merge every 2-3 lines to reassemble TOC rows
    for i in range(len(candidate_lines) - 2):
        merged = f"{candidate_lines[i]} {candidate_lines[i+1]} {candidate_lines[i+2]}"
        match = TOC_ROW_PATTERN.match(merged)
        if match:
            section = match.group(1)
            title = match.group(3).strip()
            page_num = int(match.group(4))
            toc.append({
                "section": section,
                "title": title,
                "page": page_num
            })

    doc.close()
    return toc
```

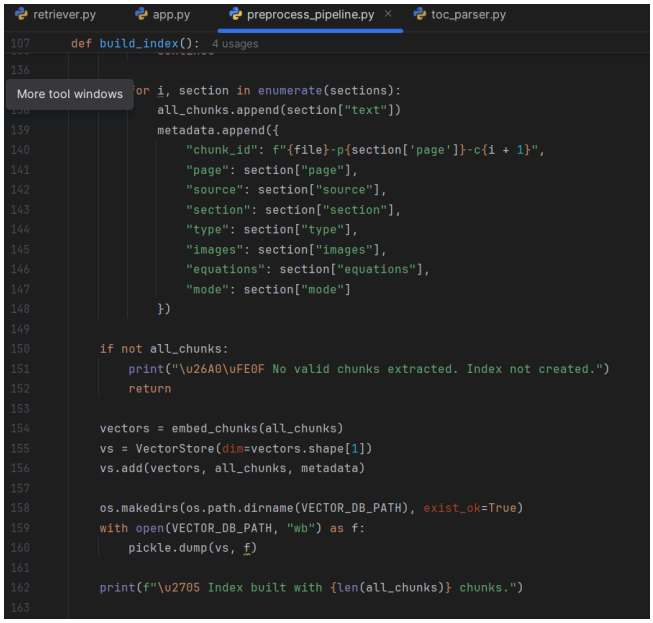
retriever.py app.py preprocess_pipeline.py toc_parser.py

```python


def extract_structured_sections(filepath, toc_lookup=None):  1 usage
    doc = fitz.open(filepath)
    chunks = []
    seen_texts = set()
    current_section = "Unknown Section"

    section_by_page = {}
    if toc_lookup:
        sorted_toc = sorted(toc_lookup, key=lambda x: x["page"])
        for i, entry in enumerate(sorted_toc):
            start = entry["page"]
            end = sorted_toc[i + 1]["page"] if i + 1 < len(sorted_toc) else 10_000
            for p in range(start, end):
                section_by_page[p] = entry["title"]

    for page_num, page in enumerate(doc, start=1):
        current_section = section_by_page.get(page_num, current_section)

        blocks = page.get_text("dict")["blocks"]
        images = [{"ext": img[1], "xref": img[0]} for img in page.get_images(full=True)]

        page_paragraphs = []
        for block in blocks:
            if "lines" not in block:
                continue

            text = " ".join(span["text"] for line in block["lines"] for span in line["spans"]).strip()
            if not text or len(text) < 10:
                continue

            # Section header detection (fallback if TOC not used)
            if SECTION_HEADER_PATTERN.match(text):
                current_section = text.strip()
                continue
```



```
retriever.py  app.py  preprocess_pipeline.py  toc_parser.py
def extract_structured_sections(filepath, toc_lookup=None):  1 usage
            if EMAIL_PATTERN.search(text.lower()):
                continue

            page_paragraphs.append(text)

        # Group paragraphs by page
        grouped = []
        current_group = []
        word_count = 0

        for para in page_paragraphs:
            wc = len(para.split())
            if para in seen_texts or wc < 10:
                continue
            seen_texts.add(para)

            current_group.append(para)
            word_count += wc

            # If group size exceeds ~300 words, commit
            if word_count >= 300:
                grouped.append(" ".join(current_group))
                current_group = []
                word_count = 0

        # Add remainder
        if current_group:
            grouped.append(" ".join(current_group))

        # Process each grouped chunk
        for para_group in grouped:
            eqs = []
            if "=" in para_group or any(w in para_group for w in ["∇", "∂", "∫", "∑", "π", "λ"]):
                eqs.append(para_group)  # crude math heuristic
```

```
retriever.py  app.py  preprocess_pipeline.py  toc_parser.py
def extract_structured_sections(filepath, toc_lookup=None):  1 usage
                    eqs.append(para_group)  # crude math heuristic

                chunks.append({
                    "text": para_group,
                    "page": page_num,
                    "source": os.path.basename(filepath),
                    "section": current_section,
                    "type": "paragraph-group",
                    "images": images,
                    "equations": eqs,
                    "mode": "page-paragraph"
                })

        doc.close()
        return chunks

TOC_GLOBAL = None  # Declare global variable for reuse

def build_index():  4 usages
    if os.path.exists(VECTOR_DB_PATH):
        os.remove(VECTOR_DB_PATH)
        print("🔁 Old vector index deleted.")

    all_chunks, metadata = [], []
    toc_titles = None

    for i, file in enumerate(sorted(os.listdir("data/raw"))):
        if not file.endswith(".pdf"):
            continue

        path = os.path.join("data/raw", file)

        # ✅ Parse TOC only once
        if toc_titles is None and "LectureNotes-1.pdf" in file:
            toc_titles = parse_toc(path)
```



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 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: Jeremi´c 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: Jeremi´c 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

```python
SECTION_PATTERN = re.compile(r"^\d+(\.\d+)*$")
PAGE_PATTERN = re.compile(r"^\d{1,4}$")

def extract_lines_from_blocks(filepath, min_page=13, max_page=89):  1 usage
    """Extracts TOC-relevant lines from pages 14 to 89, skipping headers/footers."""
    doc = fitz.open(filepath)
    lines = []

    for page_num in range(min_page, max_page):  # page 14 to 89
        page = doc[page_num]
        for block in page.get_text("blocks"):
            if len(block) >= 5:
                x0, y0, x1, y1, text = block[:5]

                #if y0 < 50 or y1 > 780:
                    #continue

                for line in text.split("\n"):
                    cleaned = line.strip()
                    if cleaned and cleaned.strip(".") != "":
                        print(f"[DEBUG] Page {page_num + 1}: {cleaned}")  # Optional: confirm line
                        lines.append(cleaned)

    doc.close()
    return lines

def parse_toc(filepath, max_pages=89):  2 usages
    """
    Parses the table of contents from the first `max_pages` of a PDF,
    returning a list of dicts: {title, page, depth}
    """
    TOC_START_PAGE = 13  # page 14 (0-indexed)
    TOC_END_PAGE = 89
    lines = extract_lines_from_blocks(filepath, min_page=TOC_START_PAGE, max_page=TOC_END_PAGE)
    toc = []
    buffer = []
```

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

```python
def parse_toc(filepath, max_pages=89):  2 usages
    for line in lines:
        buffer.append(line)

        # Try all 3-line windows in the buffer
        if len(buffer) >= 3:
            for i in range(len(buffer) - 2):
                a, b, c = buffer[i], buffer[i+1], buffer[i+2]

                if SECTION_PATTERN.match(a) and PAGE_PATTERN.match(c):
                    try:
                        section = a
                        title = b.strip().rstrip(".")  # Remove trailing dot
                        page = int(c)
                        depth = section.count(".") + 1

                        toc.append({
                            "title": f"{section} {title}",
                            "page": page,
                            "depth": depth
                        })

                        buffer = buffer[i+3:]  # Remove the matched block
                        break

                    except Exception as e:
                        print(f"⚠️ Skipping malformed TOC entry: {a}, {b}, {c} — {e}")
            else:
                # Trim runaway buffer
                if len(buffer) > 10:
                    buffer.pop(0)

    print(f"📘 TOC parsed with {len(toc)} entries.")
    if toc:
        print(f"✅ Sample TOC entry: {toc[0]}")
```

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.

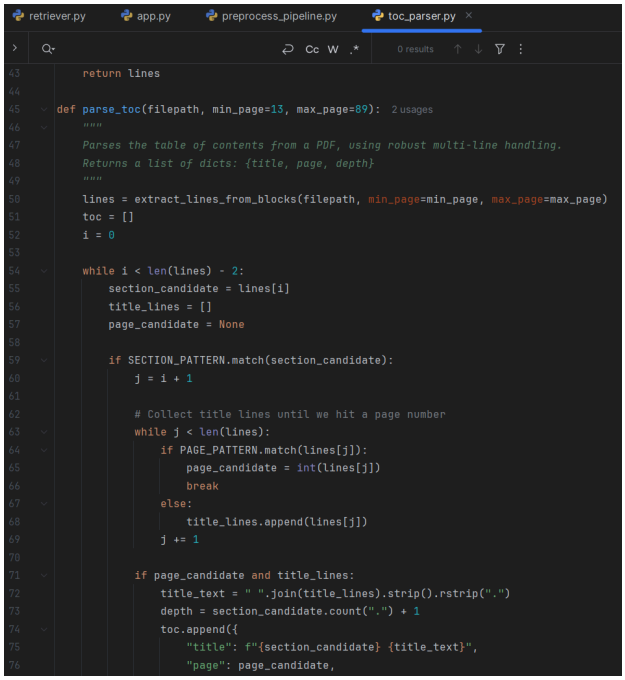
I implemented the bad line pattern sets so that the parser filters out anything with those key words, and I updated the parsing logic to accommodate for it. This let some of the titles be filtered out from the key words. Some longer titles that cover multiple lines were also messing up the parsing. It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

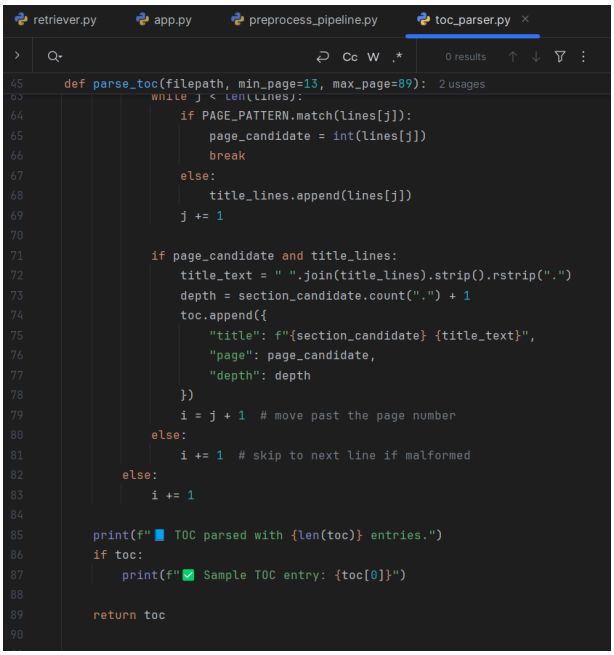
Refactor your parsing loop to:

- 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





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

```python
import re

def parse_toc_file(toc_path):  3 usages
    with open(toc_path, 'r', encoding='utf-8') as file:
        lines = file.readlines()

    entries = []
    for raw_line in lines:
        line = str(raw_line).strip()  # Ensure it's a string

        # Only process lines that start with \contentsline
        if not line.startswith(r'\contentsline'):
            continue

        # Match chapter/section lines
        match = re.search(
             pattern: r'\\contentsline\s*\{(chapter|section|subsection|subsubsection)\}'
            r'\{\s*\\numberline\s*\{([^}]*)\}\s*([^}]*)}'
            r'\{(\d+)}', line)
        if not match:
            continue

        section_type, number, title, page = match.groups()

        # Extract the correct page number just before {chapter.xxx}
        page_match = re.search( pattern: r'\{(\d+)}\{(?:chapter|section|subsection|subsubsection)\.[^}]+\}', line)

        if not page_match:
            continue
        page = int(page_match.group(1))

        # Clean title
        title = re.sub( pattern: r'\\[a-zA-Z]+\s*', repl: '', title)          # remove LaTeX commands like \relax
        title = re.sub( pattern: r'\{[^{}]*\}', repl: '', title)              # remove {...}
        title = re.sub( pattern: r'(?<![23])\b\d+\b(?![0])', repl: '', title) # Preserve "2D" and "3D" patterns, remove other stray numbers
        title = re.sub( pattern: r'[\\{}]', repl: '', title)                  # remove stray \ or braces
```

retriever.py | toc_parser.py | CompMechanicsLectureNotesTOC.toc | app.py | preprocess_pipeline.py

```python
def parse_toc_file(toc_path):  3 usages
            continue
        page = int(page_match.group(1))

        # Clean title
        title = re.sub( pattern: r'\\[a-zA-Z]+\s*', repl: '', title)          # remove LaTeX commands like \relax
        title = re.sub( pattern: r'\{[^{}]*\}', repl: '', title)              # remove {...}
        title = re.sub( pattern: r'(?<![23])\b\d+\b(?![0])', repl: '', title) # Preserve "2D" and "3D" patterns, remove other stray numbers
        title = re.sub( pattern: r'[\\{}]', repl: '', title)                  # remove stray \ or braces
        title = title.strip()

        entries.append({
            'number': number.strip(),
            'title': title,
            'page': int(page)
        })

    return entries

# Example usage (uncomment for standalone test):
# parsed = parse_toc_file("CompMechanicsLectureNotesTOC.toc")
# for entry in parsed[:10]:
#     print(entry)

# Optional test run
if __name__ == "__main__":
    toc_file = "../data/CompMechanicsLectureNotesTOC.toc"
    toc_entries = parse_toc_file(toc_file)
    for e in toc_entries[:100]:  # Display first 20 entries for inspection
        print(e)
```

Preprocess_pipeline.py:

retriever.py | toc_parser.py | CompMechanicsLectureNotesTOC.toc | app.py | preprocess_pipeline.py

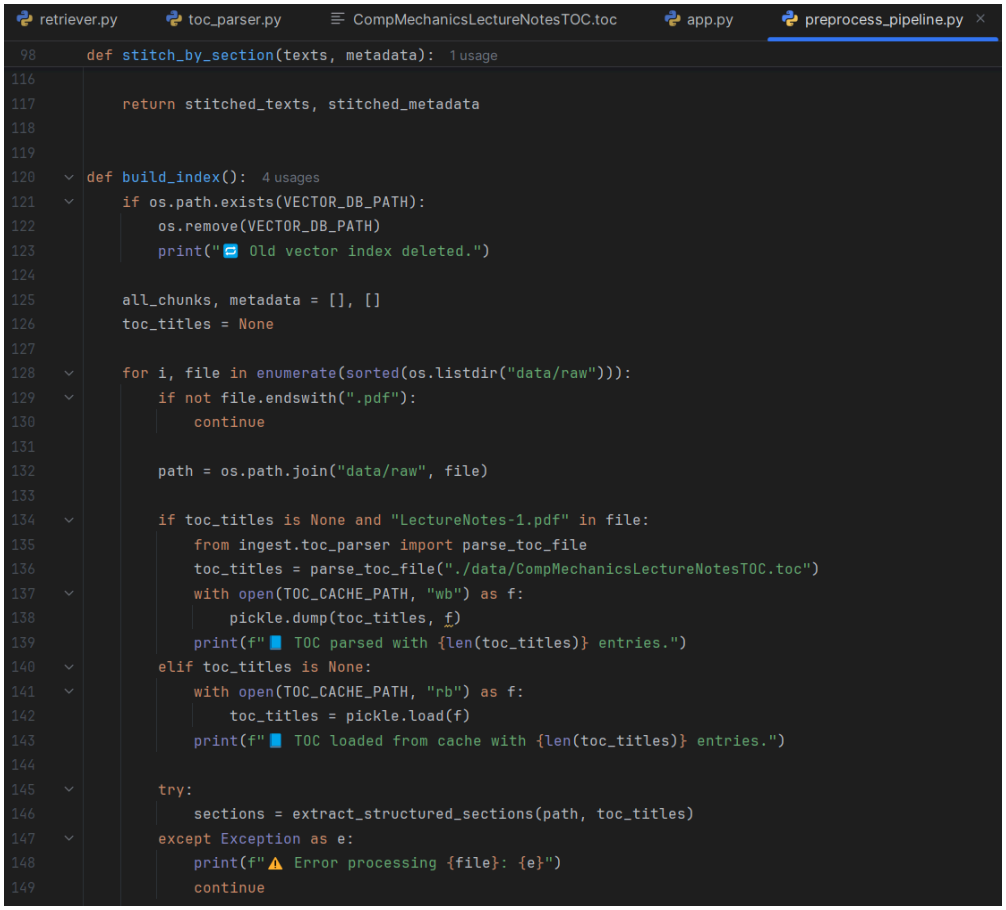
```python

def extract_structured_sections(filepath, toc_lookup=None):  1 usage
    doc = fitz.open(filepath)
    chunks = []
    seen_texts = set()
    current_section = "Unknown Section"

    section_by_page = {}
    if toc_lookup:
        sorted_toc = sorted(toc_lookup, key=lambda x: x["page"])
        for i, entry in enumerate(sorted_toc):
            start = entry["page"]
            end = sorted_toc[i + 1]["page"] if i + 1 < len(sorted_toc) else 10_000
            for p in range(start, end):
                section_by_page[p] = entry["title"]

    for page_num, page in enumerate(doc, start=1):
        current_section = section_by_page.get(page_num, current_section)

        blocks = page.get_text("dict")["blocks"]
        images = [{"ext": img[1], "xref": img[0]} for img in page.get_images(full=True)]

        page_paragraphs = []
        for block in blocks:
            if "lines" not in block:
                continue

            text = " ".join(span["text"] for line in block["lines"] for span in line["spans"]).strip()
            if not text or len(text) < 10:
                continue

            if SECTION_HEADER_PATTERN.match(text):
                current_section = text.strip()
                continue
```

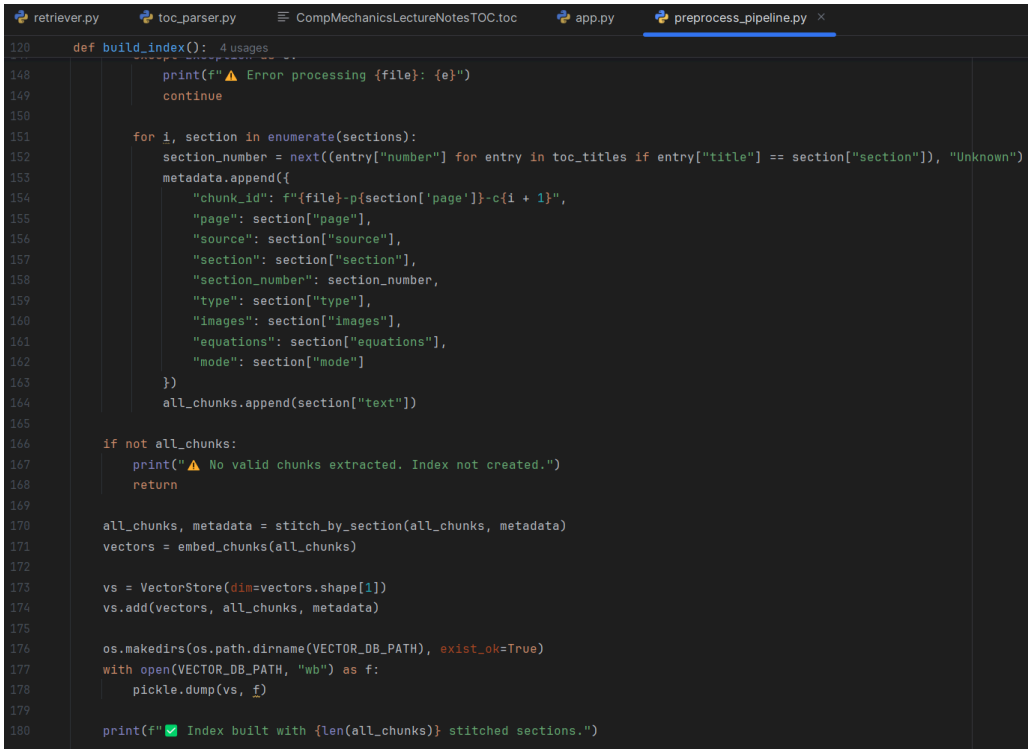
retriever.py | toc_parser.py | CompMechanicsLectureNotesTOC.toc | app.py | preprocess_pipeline.py

```python
def extract_structured_sections(filepath, toc_lookup=None):  1 usage
                continue

            if EMAIL_PATTERN.search(text.lower()):
                continue

            page_paragraphs.append(text)

        grouped = []
        current_group = []
        word_count = 0

        for para in page_paragraphs:
            wc = len(para.split())
            if para in seen_texts or wc < 10:
                continue
            seen_texts.add(para)

            current_group.append(para)
            word_count += wc

            if word_count >= 300:
                grouped.append(" ".join(current_group))
                current_group = []
                word_count = 0

        if current_group:
            grouped.append(" ".join(current_group))

        for para_group in grouped:
            eqs = []
            if "=" in para_group or any(w in para_group for w in ["∇", "∂", "→", "∫", "Σ", "π", "λ"]):
                eqs.append(para_group)

            chunks.append({
                "text": para_group,
```

retriever.py | toc_parser.py | CompMechanicsLectureNotesTOC.toc | app.py | preprocess_pipeline.py

```python
def extract_structured_sections(filepath, toc_lookup=None):  1 usage
                chunks.append({
                    "text": para_group,
                    "page": page_num,
                    "source": os.path.basename(filepath),
                    "section": current_section,
                    "type": "paragraph-group",
                    "images": images,
                    "equations": eqs,
                    "mode": "page-paragraph"
                })

    doc.close()
    return chunks


def stitch_by_section(texts, metadata):  1 usage
    from collections import defaultdict
    grouped = defaultdict(list)

    for text, meta in zip(texts, metadata):
        key = meta.get("section", "Unknown")
        grouped[key].append((text, meta))

    stitched_texts = []
    stitched_metadata = []

    for section, chunks in grouped.items():
        combined_text = "\n\n".join([chunk[0] for chunk in chunks])
        base_meta = chunks[0][1].copy()
        base_meta["stitched_chunks"] = len(chunks)
        base_meta["type"] = "stitched"
        stitched_texts.append(combined_text)
        stitched_metadata.append(base_meta)

    return stitched_texts, stitched_metadata
```





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.

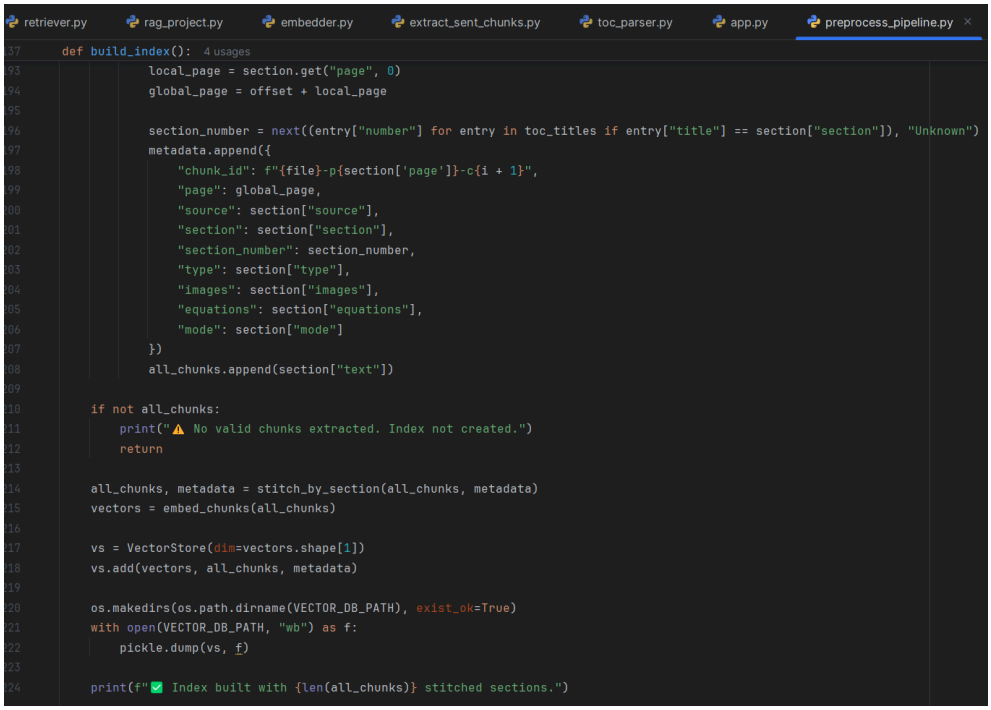
Preprocess_pipeline.py build_index() function:

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

```python
def build_index():  4 usages
    if os.path.exists(VECTOR_DB_PATH):
        os.remove(VECTOR_DB_PATH)
        print("🔁 Old vector index deleted.")

    all_chunks, metadata = [], []
    toc_titles = None

    # ✅ Mapping for page offset by file
    start_page_offset = {
        "Jeremic_et_al_CompMech_LectureNotes-1.pdf": 1,
        "Jeremic_et_al_CompMech_LectureNotes-2.pdf": 301,
        "Jeremic_et_al_CompMech_LectureNotes-3.pdf": 601,
        "Jeremic_et_al_CompMech_LectureNotes-4.pdf": 901,
        "Jeremic_et_al_CompMech_LectureNotes-5.pdf": 1201,
        "Jeremic_et_al_CompMech_LectureNotes-6.pdf": 1501,
        "Jeremic_et_al_CompMech_LectureNotes-7.pdf": 1801,
        "Jeremic_et_al_CompMech_LectureNotes-8.pdf": 2101,
        "Jeremic_et_al_CompMech_LectureNotes-9.pdf": 2401,
        "Jeremic_et_al_CompMech_LectureNotes-10.pdf": 2701,
        "Jeremic_et_al_CompMech_LectureNotes-11.pdf": 3001
    }

    for i, file in enumerate(sorted(os.listdir("data/raw"))):
        if not file.endswith(".pdf"):
            continue

        path = os.path.join("data/raw", file)

        if toc_titles is None and "LectureNotes-1.pdf" in file:
            from ingest.toc_parser import parse_toc_file
            toc_titles = parse_toc_file("./data/CompMechanicsLectureNotesTOC.toc")
            with open(TOC_CACHE_PATH, "wb") as f:
                pickle.dump(toc_titles, f)
            print(f"📘 TOC parsed with {len(toc_titles)} entries.")
        elif toc_titles is None:
```

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

```python
def build_index():  4 usages
        if toc_titles is None and "LectureNotes-1.pdf" in file:
            from ingest.toc_parser import parse_toc_file
            toc_titles = parse_toc_file("./data/CompMechanicsLectureNotesTOC.toc")
            with open(TOC_CACHE_PATH, "wb") as f:
                pickle.dump(toc_titles, f)
            print(f"📘 TOC parsed with {len(toc_titles)} entries.")
        elif toc_titles is None:
            with open(TOC_CACHE_PATH, "rb") as f:
                toc_titles = pickle.load(f)
            print(f"📘 TOC loaded from cache with {len(toc_titles)} entries.")

        # 🛠️ Filter TOC entries to match current file's local page range
        doc = fitz.open(path)
        page_count = doc.page_count
        doc.close()

        filtered_toc = [entry for entry in toc_titles if 0 <= entry["page"] < page_count]

        try:
            sections = extract_structured_sections(path, filtered_toc)
        except Exception as e:
            print(f"⚠️ Error processing {file}: {e}")
            continue

        offset = start_page_offset.get(file, 0)

        for i, section in enumerate(sections):
            local_page = section.get("page", 0)
            global_page = offset + local_page

            section_number = next((entry["number"] for entry in toc_titles if entry["title"] == section["section"]), "Unknown")
            metadata.append({
                "chunk_id": f"{file}-p{section['page']}-c{i + 1}",
                "page": global_page,
                "source": section["source"],
```



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

```python

def extract_structured_sections(filepath, toc_lookup=None):  1 usage
    doc = fitz.open(filepath)
    chunks = []
    seen_texts = set()
    current_section = "Unknown Section"

    section_by_page = {}
    if toc_lookup:
        sorted_toc = sorted(toc_lookup, key=lambda x: x["page"])
        for i, entry in enumerate(sorted_toc):
            start = entry["page"]
            end = sorted_toc[i + 1]["page"] if i + 1 < len(sorted_toc) else 10_000
            for p in range(start, end):
                section_by_page[p] = entry["title"]

    for page_num, page in enumerate(doc, start=1):
        current_section = section_by_page.get(page_num, current_section)

        blocks = page.get_text("dict")["blocks"]
        images = [{"ext": img[1], "xref": img[0]} for img in page.get_images(full=True)]

        page_paragraphs = []
        for block in blocks:
            if "lines" not in block:
                continue

            cleaned_lines = []
            for line in block["lines"]:
                line_text = " ".join(span["text"] for span in line["spans"]).strip()
                if not line_text or len(line_text) < 10:
                    continue
                if EMAIL_PATTERN.search(line_text.lower()) or AFFILIATION_PATTERN.search(line_text.lower()):
                    continue
                cleaned_lines.append(line_text)
```

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

def extract_structured_sections(filepath, toc_lookup=None): 1 usage

```python
            text = " ".join(cleaned_lines).strip()
            if not text or len(text) < 10:
                continue

            if SECTION_HEADER_PATTERN.match(text):
                current_section = text.strip()
                continue

            page_paragraphs.append(text)

        # Paragraph grouping
        grouped = []
        current_group = []
        word_count = 0

        for para in page_paragraphs:
            wc = len(para.split())
            if para in seen_texts or wc < 10:
                continue
            seen_texts.add(para)

            current_group.append(para)
            word_count += wc

            if word_count >= 300:
                grouped.append(" ".join(current_group))
                current_group = []
                word_count = 0

        if current_group:
            grouped.append(" ".join(current_group))

        for para_group in grouped:
            eqs = []
            if "=" in para_group or any(w in para_group for w in ["∇", "∂", "→", "∫", "Σ", "π", "λ"]):
```

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

```python
def extract_structured_sections(filepath, toc_lookup=None):  1 usage
        for para in page_paragraphs:
            wc = len(para.split())
            if para in seen_texts or wc < 10:
                continue
            seen_texts.add(para)

            current_group.append(para)
            word_count += wc

            if word_count >= 300:
                grouped.append(" ".join(current_group))
                current_group = []
                word_count = 0

        if current_group:
            grouped.append(" ".join(current_group))

        for para_group in grouped:
            eqs = []
            if "=" in para_group or any(w in para_group for w in ["∇", "∂", "→", "∫", "Σ", "π", "λ"]):
                eqs.append(para_group)

            chunks.append({
                "text": para_group,
                "page": page_num,
                "source": os.path.basename(filepath),
                "section": current_section,
                "type": "paragraph-group",
                "images": images,
                "equations": eqs,
                "mode": "page-paragraph"
            })

    doc.close()
    return chunks
```