Advanced Chunking Module || VINO AI for Students

- Advanced Chunking Module || VINO AI for Students
- Introduction
- Code Snippet
- Core Functionality
- Chunking Strategies
- Output Format
- Data Models
- Configuration
- Example Output
- Fallback
- Conclusion

Introduction

In this document, I will explain how advanced chunking module works. It works exceptionally well on Markdown, but for unstructured types of files (e.g.. .pdf, .docx) we must fall back to a simpler chunking technique (fixed-size token chunking).

This document will serve as a devlog, explaining how the code works and demonstrating output.

Code Snippet

```python
# chunking.py

"""
Document Chunking Module

This module provides functionality to process documents (markdown, docx, pdf)
and split them into chunks based on their table of contents structure.

Features:
- Automatic table of contents detection
- Intelligent text chunking based on document structure
- Token-aware chunk sizing with overflow handling
- Support for multiple document formats (MD, DOCX, PDF)
- Comprehensive text cleanup and normalization
"""

import os
import re
from typing import List, Tuple, Optional
from pathlib import Path

import pandas as pd
import pypandoc
import tiktoken
from dotenv import load_dotenv

from models import DocumentChunk, DocumentMetadata
from chunking_config import *

# Load environment variables
load_dotenv()

DEBUG_MODE = False  # Set to True to show debug output

def identify_doc_type(doc: str) -> str:
    """
    Categorizes a plaintext document based on the format of the table of contents.

    Args:
        doc: The document content as a string

    Returns:
        Document type classification ('TOC_WITHOUT_TITLE' or 'NO_TOC_TITLE')
    """
    return "TOC_WITHOUT_TITLE" if TOC_PATTERN.search(doc) else "NO_TOC_TITLE"


def read_doc(path: str) -> Tuple[str, str]:
    """
    Reads a document file and extracts the table of contents and full text.

    Args:
        path: File path to the document

    Returns:
        A tuple containing (table_of_contents, full_text)

    Raises:
        Exception: If the file cannot be processed by pypandoc
    """
    try:
        doc = str(pypandoc.convert_file(
            path, 'plain', format='md',
            extra_args=["--toc", "--standalone"]
        ))
        doc_type = identify_doc_type(doc)

        if doc_type == "TOC_WITH_TITLE":
            doc = re.sub(r'.*\n\n\n-', '-', doc)
            toc, text = doc.split('\n\n', 1)
        elif doc_type == "TOC_WITHOUT_TITLE":
            # Split on double newline/carriage return to separate TOC from content
            parts = re.split(r'\r?\n\r?\n', doc, 1)
            toc, text = (parts[0], parts[1]) if len(parts) >= 2 else ("", doc)
        else:
            toc, text = "", doc
```

```python
        return toc, text

    except Exception as e:
        if DEBUG_MODE:
            print(f"Error processing file {path}: {e}")
        return "", ""

def cleanup_plaintext(text: str) -> str:
    """
    Cleans up the full text of a document by normalizing whitespace and removing artifacts.

    Args:
        text: Raw text content from the document

    Returns:
        Cleaned text with normalized formatting
    """
    # Remove image artifacts and empty brackets
    text = text.replace("[image]", "").replace("[]", "")

    # Normalize line endings to \n
    text = LINE_ENDING_PATTERN.sub('\n', text)

    # Replace single \n with space EXCEPT when:
    # - followed by another \n (paragraph break)
    # - followed by "- " (bullet point)
    # - preceded by a bullet point and followed by "- " (between bullet points)
    text = NEWLINE_REPLACE_PATTERN.sub(' ', text)

    # Replace any sequence of two or more newlines with \n\n
    text = PARAGRAPH_BREAK_PATTERN.sub('\n\n', text)

    # Replace multiple spaces with single space
    text = WHITESPACE_PATTERN.sub(' ', text)

    return text

def split_text(toc: str, text: str) -> List[str]:
    """
    Splits text into chunks based on headings from the table of contents.

    Args:
        toc: Table of contents with headings
        text: Cleaned document text

    Returns:
        List of text chunks in format "Heading [SEP] Content"
    """
    # Extract headings from TOC
    headings = []
    if toc.strip():
        toc_lines = re.split(r'\r?\n', toc)
        for line in toc_lines:
            cleaned_line = line.strip('- \n\r').strip()
            if cleaned_line:
                headings.append(cleaned_line)

    paragraphs = text.split("\n\n")
    current_heading = ""
    current_content = []
    text_chunks = []

    for para in paragraphs:
        para = para.strip()
        if not para:
            continue

        # Check if this paragraph is a heading
        if headings and para in headings:
            # Save the previous heading and its content as a chunk
            if current_heading and current_content:
                combined_content = " ".join(current_content)
                text_chunks.append(f"{current_heading} [SEP] {combined_content}")

            # Start new heading
            current_heading = para
            headings.remove(para)
            current_content = []
        else:
            # Accumulate content under the current heading
            current_content.append(para)

    # Add the last heading and its content
    if current_heading and current_content:
        combined_content = " ".join(current_content)
        text_chunks.append(f"{current_heading} [SEP] {combined_content}")
    elif current_content and not current_heading:
        # Handle content without headings
        combined_content = " ".join(current_content)
        text_chunks.append(combined_content)

    return text_chunks

def process_single_file(file_path: str) -> List[DocumentChunk]:
    """
    Process a single document file and return its chunks with metadata.

    Args:
        file_path: Path to the file to process

    Returns:
        List of DocumentChunk objects containing chunk data and metadata
    """
    file_path_obj = Path(file_path)
    filename = file_path_obj.stem
    if DEBUG_MODE:
        print(f"Processing file: {file_path}")

    # Extract TOC and text
    toc, text = read_doc(file_path)
    if not text and DEBUG_MODE:
        print(f"  Warning: No text extracted from {file_path}")
        return []

    if DEBUG_MODE:
        print(f"  TOC length: {len(toc)}, Text length: {len(text)}")

    # Clean the text
    text_cleaned = cleanup_plaintext(text)
    if DEBUG_MODE:
        print(f"  Cleaned text length: {len(text_cleaned)}")

    # Split into chunks
    text_chunks = split_text(toc, text_cleaned)
    if DEBUG_MODE:
        print(f"  Number of chunks before oversized splitting: {len(text_chunks)}")
        if not text_chunks:
            print(f"    No chunks generated for {file_path}")

    # Apply oversized chunk splitting
    final_chunks = []
    for chunk in text_chunks:
        split_chunks = split_oversized_chunk(chunk, MAX_CHUNK_TOKENS)
        final_chunks.extend(split_chunks)

    if DEBUG_MODE:
        print(f"  Number of chunks after oversized splitting: {len(final_chunks)}")

    # Initialize TikToken for chunk length calculation
    encoding = tiktoken.encoding_for_model("gpt-3.5-turbo")

    # Create list of DocumentChunk objects
    document_chunks = []
    for chunk_number, chunk in enumerate(final_chunks, 1):
        tokens = encoding.encode(chunk)
        section_name = chunk.split("[SEP]")[0].strip() if "[SEP]" in chunk else "No Heading"

        # Create DocumentMetadata object
        metadata = DocumentMetadata(
            doc_id=f"{filename}_{chunk_number}",
            chunk_number=chunk_number,
            chunk_length=len(tokens),
            section=section_name
        )

        # Create DocumentChunk object
        doc_chunk = DocumentChunk(metadata=metadata, text=chunk)
        document_chunks.append(doc_chunk)

    return document_chunks

def split_oversized_chunk(chunk_text: str, max_tokens: int = MAX_CHUNK_TOKENS) ->
List[str]:
    """
    Split an oversized chunk into smaller chunks while preserving meaning.

    Args:
        chunk_text: The chunk text to split (format: "Heading [SEP] Content")
        max_tokens: Maximum tokens per chunk

    Returns:
        List of smaller chunks maintaining context
    """
    encoding = tiktoken.encoding_for_model("gpt-3.5-turbo")

    # Check if chunk needs splitting
    if len(encoding.encode(chunk_text)) <= max_tokens:
        return [chunk_text]

    # Extract heading and content
    if "[SEP]" in chunk_text:
        heading, content = chunk_text.split("[SEP]", 1)
        heading = heading.strip()
        content = content.strip()
    else:
        heading = ""
        content = chunk_text.strip()

    split_chunks = []

    # Try splitting by bullet points or numbered lists first
    if BULLET_NUMBERED_PATTERN.search(content):
        split_chunks = _split_by_list_items(content, heading, max_tokens,
encoding)
    # If no bullet points, try splitting by sentences
    elif '.' in content:
        split_chunks = _split_by_sentences(content, heading, max_tokens, encoding)
    # Last resort: split by approximate token count
    else:
        split_chunks = _split_by_words(content, heading, max_tokens)

    # If we still have oversized chunks, recursively split them
    final_chunks = []
    for chunk in split_chunks:
        if len(encoding.encode(chunk)) > max_tokens:
            final_chunks.extend(split_oversized_chunk(chunk, max_tokens))
        else:
            final_chunks.append(chunk)

    return final_chunks if final_chunks else [chunk_text]
```

```python
def _split_by_list_items(content: str, heading: str, max_tokens: int, encoding) -> List[str]:
    """Split content by bullet points or numbered items."""
    parts = BULLET_NUMBERED_PATTERN.split(content)
    current_chunk = ""
    chunks = []

    for i, part in enumerate(parts):
        if i == 0:
            current_chunk = part
        else:
            test_chunk = current_chunk + part
            test_text = f"{heading} [SEP] {test_chunk}".strip() if heading else test_chunk

            if len(encoding.encode(test_text)) > max_tokens and current_chunk.strip():
                # Save current chunk and start new one
                final_text = f"{heading} [SEP] {current_chunk}".strip() if heading else current_chunk.strip()
                chunks.append(final_text)
                current_chunk = part if i + 1 < len(parts) else ""
            else:
                current_chunk = test_chunk

    # Add remaining content
    if current_chunk.strip():
        final_text = f"{heading} [SEP] {current_chunk}".strip() if heading else current_chunk.strip()
        chunks.append(final_text)

    return chunks


def _split_by_sentences(content: str, heading: str, max_tokens: int, encoding) -> List[str]:
    """Split content by sentences."""
    sentences = SENTENCE_SPLIT_PATTERN.split(content)
    current_chunk = ""
    chunks = []

    for sentence in sentences:
        test_chunk = f"{current_chunk} {sentence}".strip() if current_chunk else sentence
        test_text = f"{heading} [SEP] {test_chunk}".strip() if heading else test_chunk

        if len(encoding.encode(test_text)) > max_tokens and current_chunk.strip():
            # Save current chunk and start new one
            final_text = f"{heading} [SEP] {current_chunk}".strip() if heading else current_chunk.strip()
            chunks.append(final_text)
            current_chunk = sentence
        else:
            current_chunk = test_chunk

    # Add remaining content
    if current_chunk.strip():
        final_text = f"{heading} [SEP] {current_chunk}".strip() if heading else
current_chunk.strip()
        chunks.append(final_text)

    return chunks


def _split_by_words(content: str, heading: str, max_tokens: int) -> List[str]:
    """Split content by approximate word count."""
    words = content.split()
    # Rough estimate: 1 token ≈ 0.75 words
    words_per_chunk = int(max_tokens * 0.75)
    chunks = []

    for i in range(0, len(words), words_per_chunk):
        chunk_words = words[i:i + words_per_chunk]
        chunk_content = " ".join(chunk_words)
        final_text = f"{heading} [SEP] {chunk_content}".strip() if heading else
chunk_content.strip()
        chunks.append(final_text)

    return chunks

def process_documents(root_dir: str = ROOT_DIR,
                      allowed_filetypes: List[str] = ALLOWED_FILETYPES) ->
List[DocumentChunk]:
    """
    Process all documents in a directory and return a list of chunks.

    Args:
        root_dir: Root directory to search for documents
        allowed_filetypes: List of allowed file extensions

    Returns:
        List of DocumentChunk objects containing all processed chunks
    """
    if DEBUG_MODE:
        print(f"Walking directory: {os.path.abspath(root_dir)}")

    all_chunk_data = []
    processed_files = 0

    for directory, subdirectories, files in os.walk(root_dir):
        if DEBUG_MODE:
            print(f"In directory: {directory}")

        for file in files:
            filename, filetype = os.path.splitext(file)
            if filetype in allowed_filetypes:
                full_path = os.path.join(directory, file)
                try:
                    chunk_data = process_single_file(full_path)
                    all_chunk_data.extend(chunk_data)
                    processed_files += 1
                except Exception as e:
                    print(f"Error processing file {full_path}: {e}")
            elif DEBUG_MODE:
                print(f"Skipping file (wrong type): {os.path.join(directory, file)}")

    print(f"Processed {processed_files} files, created {len(all_chunk_data)} chunks")

    if DEBUG_MODE:
        _print_debug_chunks(all_chunk_data)

    return all_chunk_data


def _print_debug_chunks(chunks: List[DocumentChunk]) -> None:
    """Print detailed information about chunks for debugging."""
    for i, chunk in enumerate(chunks, 1):
        print(f"\n{'='*60}")
        print(f"CHUNK {i}")
        print(f"{'='*60}")
        print(f"Doc ID: {chunk.metadata.doc_id}")
        print(f"Section: {chunk.metadata.section}")
        print(f"Chunk Number: {chunk.metadata.chunk_number}")
        print(f"Token Length: {chunk.metadata.chunk_length}")
        print(f"{'-'*60}")
        print("TEXT:")
        print(f"{'-'*60}")
        print(chunk.text)
        print(f"{'='*60}\n")

def main() -> None:
    """
    Main function to process documents and display results.
    """
    try:
        result = process_documents()
        print("\n" + "="*50)
        print("PROCESSING COMPLETE")
        print("="*50)
        print(f"Successfully processed {len(result)} chunks")

    except Exception as e:
        print(f"Error in main processing: {e}")
        raise
```

```
if __name__ == "__main__":
    main()
```

Core Functionality

Document Processing Workflow

1. File Reading (read_doc): Uses pypandoc to convert documents to plain text with TOC extraction
2. Content Cleanup (cleanup_plaintext): Normalizes whitespace and removes formatting artifacts
3. Structure-based Splitting (split_text): Uses TOC headings to create logical chunks
4. Overflow Handling (split_oversized_chunk): Further splits chunks that exceed token limits

Chunking Strategies

The module employs a hierarchical approach to handle oversized chunks:

1. List-based Splitting: Combines entire bullet point section to main context
2. Sentence-based Splitting: Breaks content at sentence boundaries
3. Word-based Splitting: Final fallback using approximate token-to-word ratios

Output Format

Each chunk is formatted as:

`Heading [SEP] Content`

This format preserves context by including the relevant heading with each content section for easy retrieval from vector store.

Data Models

Existing data models needed to be updated to complement the new metadata structure:

- **DocumentChunk**: Contains chunk text and metadata
- **DocumentMetadata**: Stores document ID, chunk number, token length, and section information

Configuration

The module uses external configuration (chunking_config.py) for:

- Maximum token limits per chunk
- File type allowlists
- Text processing patterns (regex)
- Directory paths

Example Output

To demonstrate the output, I've made this program debug_chunking.py. This is its output:

```
================================================================================
DEBUGGING FILE: kb/CMD.md
================================================================================

ORIGINAL FILE PREVIEW (50 tokens):
--------------------------------------------------
CMD Research Methods

Summary

Creative Media and Design (CMD) research methods provide a systematic and user-centered approach to understanding user needs, exploring design opportunities, creating innovative solutions, and evaluating their effectiveness. These methods are integral to developing digital products, services...

CHUNKS (19 total):
==================================================

Chunk 1:
  Doc ID: CMD_1
  Section: Summary
  Tokens: 123
  Preview (20 tokens): Summary [SEP] Creative Media and Design (CMD) research methods provide a systematic and user-centered approach...
------------------------------

Chunk 2:
  Doc ID: CMD_2
  Section: Full Description: Understanding CMD Research
  Tokens: 165
  Preview (20 tokens): Full Description: Understanding CMD Research [SEP] CMD research is a multifaceted discipline focused on investigating...
------------------------------

Chunk 3:
  Doc ID: CMD_3
  Section: Core Principles & Philosophy
  Tokens: 296
  Preview (20 tokens): Core Principles & Philosophy [SEP] The philosophy underpinning CMD research is deeply rooted in a human...
------------------------------

Chunk 4:
  Doc ID: CMD_4
  Section: Core Principles & Philosophy
  Tokens: 66
  Preview (20 tokens): Core Principles & Philosophy [SEP] Bias awareness and
```

```
mitigation are crucial. - Impactful & Actionable...
------------------------------

Chunk 5:
  Doc ID: CMD_5
  Section: The CMD Research Process
  Tokens: 289
  Preview (20 tokens): The CMD Research Process [SEP] While specific research projects may vary, the CMD research process is generally...
------------------------------

Chunk 6:
  Doc ID: CMD_6
  Section: The CMD Research Process
  Tokens: 243
  Preview (20 tokens): The CMD Research Process [SEP] Prototype & Test (or Build & Evaluate): - Goal: To...
------------------------------

Chunk 7:
  Doc ID: CMD_7
  Section: Key CMD Research Methods
  Tokens: 56
  Preview (20 tokens): Key CMD Research Methods [SEP] CMD practitioners have a vast toolkit of methods. The choice of method...
------------------------------
```

You can see a preview of the original file (50 tokens), and a preview of the first 7 chunks (it was 19 in total).

Fallback

The drawback of this modules is that it does a very poor job at chunking .pdfs and .docx due to their unstructred nature. Hence, I used simple token-based fixed-size chunking technique which can be found at document_processor.py by the function name `_process_with_fixed_size_chunking()`.

Conclusion

In the advanced chunking module we successfully implemented a structure-aware approach to document processing. By maintaining document hierarchy through TOC headings and employing a hierarchical splitting strategy (list-based → sentence-based → word-based), the module creates contextually meaningful chunks with comprehensive metadata integration. The `[SEP]` separator format preserves heading context for accurate vector database retrieval, while the fallback mechanism to fixed-size chunking ensures reliable processing across all document formats.

Future development should focus on improving unstructured document handling through advanced PDF parsing techniques or machine learning-based structure detection.