

Document Fragmentation System

Overview

A system that intelligently splits PDF/text documents into semantically coherent fragments using local LLMs (Llama/Phi) via Ollama. Each fragment maintains contextual completeness while adhering to specified size constraints.

System Architecture

Core Components

- 1. Document Processor** - Extracts text from PDF/TXT files
- 2. Fragment Analyzer** - Uses LLM to identify semantic boundaries
- 3. Size Manager** - Ensures fragments meet target size requirements
- 4. Ollama Interface** - Communicates with local LLM models

Technology Stack

- LLM Backend:** Ollama (Llama/Phi models)
- PDF Processing:** PyPDF2 or pdfplumber
- Language:** Python 3.8+
- Dependencies:** requests, PyPDF2/pdfplumber, python-dotenv

Installation & Setup

Prerequisites

```
bash

# Install Ollama on macOS
brew install ollama

# Pull required models
ollama pull llama3.1
ollama pull phi3
```

Python Dependencies

```
bash

pip install requests PyPDF2 python-dotenv
# Alternative for better PDF handling:
pip install pdfplumber
```

Implementation Design

1. Document Input Handler

```
python

class DocumentProcessor:
    def extract_text(self, file_path):
        # Handle PDF and TXT files
        # Return clean text string

    def preprocess_text(self, text):
        # Clean and normalize text
        # Remove excessive whitespace, fix encoding issues
```

2. Fragment Engine

```
python

class FragmentEngine:
    def __init__(self, model_name="llama3.1", target_size=500):
        self.model = model_name
        self.target_size = target_size

    def create.fragments(self, text):
        # Main fragmentation logic
        # Returns list of semantically complete fragments
```

3. LLM Integration

```
python

class OllamaClient:
    def __init__(self, base_url="http://localhost:11434"):
        self.base_url = base_url

    def generate_response(self, prompt, model):
        # Send requests to Ollama API
        # Handle response parsing
```

Fragmentation Strategy

Semantic Boundary Detection

The system will use the LLM to identify natural breaking points by:

1. Paragraph Analysis - Detecting topic shifts

2. Sentence Completion - Ensuring fragments end at complete thoughts

3. Context Preservation - Maintaining logical flow between fragments

Size Management

- **Target Size:** User-defined character/word count
- **Flexibility Range:** ±20% of target size for semantic completeness
- **Minimum Fragment Size:** 100 characters (configurable)
- **Maximum Fragment Size:** 150% of target size

Prompt Engineering

Fragment Identification Prompt

Analyze the following text and identify natural breaking points where the content can be split into semantically complete fragments. Each fragment should:

- Be approximately {target_size} characters long
- End at a complete thought or logical conclusion
- Maintain context independence where possible

Text: {input_text}

Return the split positions as character indices.

Fragment Validation Prompt

Evaluate if this text fragment is semantically complete and coherent:

{fragment_text}

Rate completeness (1-10) and suggest improvements if needed.

Usage Examples

Basic Usage

python

```
from document_fragmenter import DocumentFragmenter

# Initialize system
fragmenter = DocumentFragmenter(
    model="llama3.1",
    target_size=800
)

# Process document
fragments = fragmenter.process_file("document.pdf")

# Access fragments
for i, fragment in enumerate(fragments):
    print(f"Fragment {i+1}: {len(fragment)} characters")
    print(fragment[:100] + "...")
```

Advanced Configuration

```
python

# Custom settings
fragmenter = DocumentFragmenter(
    model="phi3",
    target_size=1200,
    flexibility=0.25, # ±25% size variance
    min_fragment_size=200,
    overlap_sentences=1 # Overlap for context
)
```

Configuration Options

Model Selection

- **Llama3.1:** Better for complex documents, slower processing
- **Phi3:** Faster processing, suitable for simpler texts

Size Parameters

- `target_size`: Target fragment size in characters
- `flexibility`: Allowed size variance (0.0-1.0)
- `min_fragment_size`: Minimum acceptable fragment size
- `max_fragment_size`: Maximum acceptable fragment size

Processing Options

- `overlap_sentences`: Number of sentences to overlap between fragments
- `preserve_paragraphs`: Prefer paragraph boundaries for splits
- `context_window`: Characters to consider for semantic analysis

Output Format

Fragment Structure

```
python

{
    "fragment_id": int,
    "text": str,
    "start_position": int,
    "end_position": int,
    "size": int,
    "completeness_score": float,
    "semantic_boundaries": list
}
```

Export Options

- **JSON**: Structured data with metadata
- **Plain Text**: Simple text files numbered sequentially
- **CSV**: Tabular format for analysis

Performance Considerations

Optimization Strategies

1. **Batch Processing**: Process multiple documents simultaneously
2. **Caching**: Store LLM responses for similar text patterns
3. **Parallel Processing**: Use multiple Ollama instances
4. **Memory Management**: Stream large documents instead of loading entirely

Expected Performance

- **Small Documents** (< 10 pages): 30-60 seconds
- **Medium Documents** (10-50 pages): 2-5 minutes
- **Large Documents** (50+ pages): 5-15 minutes

Error Handling

Common Issues

1. **PDF Extraction Errors:** Fallback to OCR or manual text input
2. **LLM Timeout:** Retry with smaller chunks
3. **Fragment Size Violations:** Adjust flexibility parameters
4. **Incomplete Fragments:** Manual review and adjustment

Validation

- Check fragment completeness scores
- Verify size constraints
- Ensure no content loss during processing

Testing Strategy

Unit Tests

- Text extraction accuracy
- Fragment size compliance
- Semantic boundary detection
- LLM response handling

Integration Tests

- End-to-end document processing
- Multiple file format support
- Error recovery mechanisms

Future Enhancements

Potential Improvements

1. **Multi-language Support:** Handle non-English documents
2. **Custom Semantic Rules:** User-defined fragmentation criteria
3. **Visual Processing:** Handle images and tables in PDFs
4. **Batch API:** Process multiple documents via REST API
5. **GUI Interface:** User-friendly document upload and processing

Advanced Features

- **Fragment Relationships:** Track connections between fragments
- **Topic Modeling:** Automatic categorization of fragments
- **Quality Metrics:** Automated fragment quality assessment
- **Export Integrations:** Direct export to various platforms

Troubleshooting

Common Problems

- 1. Ollama Connection Issues:** Check service status and port
- 2. Model Loading Errors:** Verify model availability
- 3. Memory Issues:** Reduce batch size or fragment target size
- 4. Encoding Problems:** Ensure UTF-8 text processing

Debug Mode

Enable verbose logging to track:

- LLM request/response cycles
- Fragment boundary decisions
- Performance metrics
- Error stack traces

Conclusion

This system provides a robust foundation for intelligent document fragmentation using local LLMs. The modular design allows for easy customization and extension based on specific requirements.