Contents

1 PDF Section Extractor - Text Attachment Support

1.1 Overview

The `PDFSectionExtractor` now supports both PDF and plain text (.txt) attachments from CouchDB. Text files are processed by replacing form feed characters (^L) with page markers, making them compatible with the same section extraction pipeline as PDFs.

Date: 2025-12-22 **Breaking Changes**: None (backward compatible)

1.2 Features

1.2.1 Text File Support

- **Automatic Detection**: Finds .txt attachments if no PDF is present
- **Form Feed Replacement**: Converts `\f` (form feed, ASCII 12) to page markers
- **Compatible Processing**: Uses same DataFrame schema and parsing as PDFs
- **YEDDA Support**: Full support for YEDDA annotations in text files
- **Encoding Handling**: UTF-8 with Latin-1 fallback

1.2.2 Attachment Priority

When `attachment_name` is not specified: 1. **First**: Search for PDF attachments (`.pdf` extension or `application/pdf` content type) 2. **Second**: Search for text attachments (`.txt` extension or `text/*` content type) 3. **Error**: Raise ValueError if neither found

1.3 Usage

1.3.1 Basic Example - Auto-Detection

```python
from pyspark.sql import SparkSession
from pdf_section_extractor import PDFSectionExtractor

spark = SparkSession.builder.appName("Extractor").getOrCreate()
extractor = PDFSectionExtractor(spark=spark)

# Automatically finds PDF or .txt attachment
df = extractor.extract_from_document(
    database='mydb',
    doc_id='doc123'
)

# Works identically for both PDFs and text files
df.show()
```

1.3.2 Explicit Text File

```python
# Specify .txt attachment explicitly
df = extractor.extract_from_document(
    database='mydb',
    doc_id='doc123',
    attachment_name='article.txt'
)
```

1.4 Form Feed Processing

1.4.1 Input Format

Text files with form feed characters (used for page breaks):

```
Page 1 content
More page 1 text
^L
Page 2 content
More page 2 text
^L
Page 3 content
```

Where ^L represents the form feed character (ASCII 12, `\f` in Python).

1.4.2 Output Format

After processing, form feeds become page markers:

```
--- PDF Page 1 ---
Page 1 content
More page 1 text

--- PDF Page 2 ---
Page 2 content
More page 2 text

--- PDF Page 3 ---
Page 3 content
```

This format is identical to PDF extraction, enabling unified processing.

1.5 DataFrame Output

Text files produce the same DataFrame schema as PDFs:

Column	Type	Description
value	String	Section/paragraph text
doc_id	String	Document ID
attachment_name	String	Attachment filename
paragraph_number	Integer	Sequential paragraph number
line_number	Integer	Line number of first line
page_number	Integer	Page number (from markers)
empirical_page_number	Integer (nullable)	Extracted page number
section_name	String (nullable)	Section name
label	String (nullable)	YEDDA annotation label

1.6 Complete Example

1.6.1 Text File with YEDDA Annotations

```
[@ Introduction section
This is the introduction.
#Introduction*]

Some regular text.
^L
[@ Methods section
This describes methods.
#Methods*]

More content.
^L
Conclusion text.
```

1.6.2 Processing

```
from pyspark.sql import SparkSession
from pdf_section_extractor import PDFSectionExtractor

spark = SparkSession.builder.appName("TextExtraction").getOrCreate()
extractor = PDFSectionExtractor(spark=spark, verbosity=2)

# Extract sections
df = extractor.extract_from_document(
    database='research_docs',
    doc_id='article_001',
    attachment_name='article.txt'
)

# Query by page
page1 = df.filter(df.page_number == 1)
page1.select("value", "label").show()

# Query by label
intro = df.filter(df.label == "Introduction")
intro.show()

# Get statistics
print(f"Total sections: {df.count()}")
print(f"Pages: {df.select('page_number').distinct().count()}")
```

1.6.3 Output

```
Retrieved attachment: article.txt (234 bytes)
Extracted 310 characters from text file (3 pages)
Parsed 5 sections/paragraphs

Total sections: 5
Pages: 3
```

1.7 File Type Detection

1.7.1 By Extension

```
# Explicit by extension
if attachment_name.endswith('.pdf'):
    # PDF processing
elif attachment_name.endswith('.txt'):
    # Text processing
```

1.7.2 By Content

For ambiguous cases (no extension), content-based detection:

```
if file_data[:4] == b'%PDF':
    # PDF magic bytes detected
else:
    # Assume text file
```

1.8 Integration with SkolClassifierV2

Text files work seamlessly with section-mode classification:

```
from skol_classifier.classifier_v2 import SkolClassifierV2

# Classifier will handle both PDFs and .txt files
classifier = SkolClassifierV2(
    spark=spark,
    input_source='couchdb',
    couchdb_url='http://localhost:5984',
    couchdb_database='mixed_documents',
    extraction_mode='section',
    use_suffixes=True
)

# Auto-discovers both PDFs and text files with form feeds
sections_df = classifier.load_raw()
classifier.fit(sections_df)
```

1.9 Benefits

1.9.1 1. Unified Processing

- Same DataFrame schema for PDFs and text
- Same parsing logic and section detection
- Same YEDDA annotation support

1.9.2 2. Flexibility

- Process legacy text files with form feeds
- Mix PDFs and text files in same database
- Seamless fallback if PDF extraction fails

1.9.3 3. Simplicity

- No code changes needed to switch between formats

- Automatic file type detection
- Compatible with all existing tools

1.10 Character Encoding

1.10.1 UTF-8 Default

Text files are decoded as UTF-8 by default:

```
text = txt_data.decode('utf-8')
```

1.10.2 Latin-1 Fallback

If UTF-8 decoding fails, falls back to Latin-1 with error replacement:

```
except UnicodeDecodeError:
    text = txt_data.decode('latin-1', errors='replace')
```

This handles most text encodings gracefully.

1.11 Form Feed Characters

1.11.1 What is Form Feed?

- **ASCII**: 12 (0x0C)
- **Escape**: \f
- **Ctrl**: Ctrl+L
- **Unicode**: U+000C
- **Purpose**: Historical page break character

1.11.2 Common Sources

- Legacy text processing systems
- OCR output from scanned documents
- Text exports from PDF converters
- Mainframe/terminal output

1.11.3 Visual Representation

In many editors, form feed appears as ^L or a special symbol.

1.12 Limitations

1.12.1 Text File Constraints

1. **No Font Information**: Unlike PDFs, text files have no formatting

2. **Manual Page Breaks**: Requires form feed characters for page numbering
3. **No Layout**: Text files don't preserve spatial layout
4. **Section Detection**: Relies on text patterns, not PDF structure

1.12.2 Not Supported

- Text files without form feeds (treated as single page)
- Binary file formats other than PDF
- Rich text formats (RTF, DOCX, etc.)
- HTML or XML documents

1.13 Examples

1.13.1 Example 1: Converting Legacy Data

```
# Legacy system exported with form feeds
df = extractor.extract_from_document(
    database='legacy_archive',
    doc_id='report_1985_123'
)

# Now accessible with modern tools
df.write.parquet('modernized_data.parquet')
```

1.13.2 Example 2: Mixed Document Types

```
# Database with both PDFs and text files
from pdf_section_extractor import PDFSectionExtractor

extractor = PDFSectionExtractor(spark=spark)

# Process all documents (auto-detects type)
doc_ids = ['doc1', 'doc2', 'doc3']  # Mix of PDFs and .txt

all_sections = []
for doc_id in doc_ids:
    df = extractor.extract_from_document('mydb', doc_id)
    all_sections.append(df)

# Union all sections
from functools import reduce
combined_df = reduce(lambda a, b: a.union(b), all_sections)

print(f"Total sections from all documents: {combined_df.count()}")
```

1.13.3 Example 3: Form Feed Analysis

```
# Check how many pages in a text file
text_data = db.get_attachment('doc123', 'article.txt').read()
text = text_data.decode('utf-8')

form_feeds = text.count('\f')
pages = form_feeds + 1

print(f"Text file has {form_feeds} form feeds = {pages} pages")
```

1.14 Testing

1.14.1 Test Suite

Run the text attachment test suite:

```
python test_txt_attachment.py
```

Tests cover: - Form feed replacement - Page marker insertion - DataFrame schema compatibility - YEDDA annotation parsing - Page number tracking - Multi-page documents

1.14.2 Manual Testing

```
# Create test text with form feeds
test_text = "Page 1\nMore text\fPage 2\nMore text\fPage 3"
test_bytes = test_text.encode('utf-8')

# Process
extractor = PDFSectionExtractor(spark=spark, verbosity=2)
processed = extractor.txt_to_text_with_pages(test_bytes)

print(processed)
# Should show:
# --- PDF Page 1 ---
# Page 1
# More text
#
# --- PDF Page 2 ---
# Page 2
# More text
#
# --- PDF Page 3 ---
# Page 3
```

1.15 Implementation Details

1.15.1 Code Changes

File: `pdf_section_extractor.py`

Changes: 1. Updated `find_pdf_attachment()` - now searches for .txt files (lines 151-182) 2. Added `txt_to_text_with_pages()` - processes text with form feeds (lines 230-271) 3. Updated `extract_from_document()` - handles both file types (lines 836-914)

1.15.2 Methods

1.15.2.1 `txt_to_text_with_pages(txt_data: bytes) -> str`

Processes text files by: 1. Decoding bytes to string (UTF-8, fallback to Latin-1) 2. Splitting on form feed characters (`\f`) 3. Adding page markers between pages 4. Ensuring proper spacing

1.15.2.2 `find_pdf_attachment()` - Enhanced

Search order: 1. PDFs (`.pdf` extension or `application/pdf` content type) 2. Text files (`.txt` extension or `text/*` content type) 3. Return None if neither found

1.15.2.3 `extract_from_document()` - Enhanced

File type detection: 1. Check extension (`.pdf` vs `.txt`) 2. Check content (PDF magic bytes %PDF) 3. Route to appropriate processor

1.16 Backward Compatibility

1.16.1 Fully Compatible

- ✅ Existing PDF processing unchanged
- ✅ New functionality is additive only
- ✅ Same DataFrame schema
- ✅ No API changes required

1.16.2 Migration

None required. Existing code works without modification.

1.17 Future Enhancements

1.17.1 Potential Improvements

1. **More Formats**: Support for RTF, DOCX, HTML

2. **Page Detection**: Heuristic page break detection without form feeds
3. **Encoding Auto-Detection**: Automatic encoding detection
4. **Markdown Support**: Parse markdown headers as sections
5. **Line Numbering**: Optional preservation of original line numbers

1.18 See Also

- PDF_SECTION_EXTRACTOR_SUMMARY.md - Complete feature overview
- PDF_YEDDA_ANNOTATION_SUPPORT.md - YEDDA annotations
- CLASSIFIER_V2_TOKENIZER_UPDATE.md - Classifier integration

Status: ✅ Complete and Tested **Version**: Added 2025-12-22 **Breaking Changes**: None **New Features**: - ✅ Text file (.txt) attachment support - ✅ Form feed character replacement - ✅ Automatic file type detection - ✅ UTF-8 and Latin-1 encoding support - ✅ Full YEDDA annotation support in text files - ✅ Identical DataFrame schema as PDFs

Known Limitations: - Requires form feed characters for page breaks - No formatting preservation - Text-only processing (no images, tables, etc.)