Contents

1 PDF Section Extractor - DataFrame Output

1.1 Overview

The `PDFSectionExtractor` has been updated to return **PySpark DataFrames** instead of simple lists of strings. This provides rich metadata and powerful querying capabilities for PDF section extraction.

1.2 What Changed

1.2.1 Before (List-based output)

```
extractor = PDFSectionExtractor()
sections = extractor.extract_from_document(
    database='skol_dev',
    doc_id='document-id'
)
# Returns: ['section 1', 'section 2', ...]
```

1.2.2 After (DataFrame-based output)

```
from pyspark.sql import SparkSession

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

sections_df = extractor.extract_from_document(
    database='skol_dev',
    doc_id='document-id'
)
# Returns: PySpark DataFrame with rich metadata
```

1.3 DataFrame Schema

The returned DataFrame has the following columns:

Column	Type	Description
`value`	string	Section/paragraph text content
`doc_id`	string	CouchDB document ID
`attachment_name`	string	Name of the PDF attachment
`paragraph_number`	integer	Sequential paragraph number (1-indexed)
`line_number`	integer	Line number of first line in the section
`page_number`	integer	PDF page number (extracted from page markers)

1.3.1 Example Output

```
root
 |-- value: string (nullable = false)
 |-- doc_id: string (nullable = false)
```

```
 |-- attachment_name: string (nullable = false)
 |-- paragraph_number: integer (nullable = false)
 |-- line_number: integer (nullable = false)
 |-- page_number: integer (nullable = false)
```

1.4 Key Features

1.4.1 1. Rich Metadata

Every section includes contextual information: - Which document it came from - Which PDF attachment - Location within the PDF (page and line numbers) - Sequential ordering (paragraph number)

1.4.2 2. Powerful Querying

Use Spark SQL or DataFrame API for complex queries:

```
# Get all sections from page 1
page1 = sections_df.filter(sections_df.page_number == 1)

# Search for keywords
results = sections_df.filter(
    sections_df.value.contains("methodology")
)

# SQL queries
sections_df.createOrReplaceTempView("sections")
spark.sql("""
    SELECT paragraph_number, value, page_number
    FROM sections
    WHERE LOWER(value) LIKE '%ascospores%'
    ORDER BY paragraph_number
""")
```

1.4.3 3. Aggregation & Analytics

Compute statistics across sections:

```
# Count sections per page
sections_df.groupBy("page_number").count().show()

# Average section length per page
from pyspark.sql.functions import length, avg
sections_df.groupBy("page_number") \
    .agg(avg(length("value")).alias("avg_length")) \
    .show()
```

1.4.4 4. Export Options

Convert to various formats:

```
# To Pandas
pandas_df = sections_df.toPandas()

# To JSON
sections_df.write.json("output.json")

# To Parquet
sections_df.write.parquet("output.parquet")

# To CSV
sections_df.write.csv("output.csv", header=True)
```

1.5 Usage Examples

1.5.1 Basic Extraction

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

# Initialize Spark
spark = SparkSession.builder \
    .appName("PDFExtractor") \
    .getOrCreate()

# Initialize extractor with Spark
extractor = PDFSectionExtractor(spark=spark)

# Extract sections
sections_df = extractor.extract_from_document(
    database='skol_dev',
    doc_id='00df9554e9834283b5e844c7a994ba5f'
)

# View schema
sections_df.printSchema()

# Show first 5 sections
sections_df.select("paragraph_number", "value", "page_number").show(5)
```

1.5.2 Multiple Documents

```
# Extract from multiple documents
doc_ids = ['doc1', 'doc2', 'doc3']

combined_df = extractor.extract_from_multiple_documents(
    database='skol_dev',
    doc_ids=doc_ids
)

# All sections from all documents in one DataFrame
print(f"Total sections: {combined_df.count()}")

# Group by document
combined_df.groupBy("doc_id").count().show()
```

1.5.3 Advanced Queries

```
# Find introduction sections
intro = sections_df.filter(
    sections_df.value.rlike("(?i)^introduction$")
)

# Get sections between line 100 and 200
mid_sections = sections_df.filter(
    (sections_df.line_number >= 100) &
    (sections_df.line_number <= 200)
)

# Find longest sections per page
from pyspark.sql.functions import length, max

sections_df.groupBy("page_number") \
    .agg(max(length("value")).alias("max_length")) \
    .orderBy("page_number") \
    .show()
```

1.5.4 Working with Metadata

```
# Get unique document IDs
doc_ids = sections_df.select("doc_id").distinct().collect()

# Count paragraphs per document
sections_df.groupBy("doc_id", "attachment_name") \
    .count() \
    .show()
```

```
# Find sections on last page
max_page = sections_df.agg({"page_number": "max"}).collect()[0][0]
last_page = sections_df.filter(sections_df.page_number == max_page)
```

1.6 Implementation Details

1.6.1 Page Number Tracking

Page numbers are extracted from PDF page markers in the text:

```
--- PDF Page 1 ---
<content>
--- PDF Page 2 ---
<content>
```

The `_get_pdf_page_marker()` method uses regex to extract page numbers:

```
def _get_pdf_page_marker(self, line: str):
    """Extract page number from PDF page marker."""
    pattern = r'^---\s*PDF\s+Page\s+(\d+)\s*---\s*$'
    return re.match(pattern, line.strip())
```

1.6.2 Line Number Tracking

Line numbers are 1-indexed and track the first line of each section/paragraph:

```
for i, line in enumerate(lines):
    line_number = i + 1  # 1-indexed

    # For headers: use current line number
    # For paragraphs: use first line of paragraph
    if current_paragraph_start_line is None:
        current_paragraph_start_line = line_number
```

1.6.3 Paragraph Numbering

Paragraphs are numbered sequentially within each document:

```
paragraph_number = 0

# Each time a section is added:
paragraph_number += 1
records.append({
    'paragraph_number': paragraph_number,
```

```
    # ...
})
```

1.7 Migration Guide

1.7.1 For Existing Code

If you have existing code using the list-based API, you need to:

1. **Initialize Spark**:

```
from pyspark.sql import SparkSession

spark = SparkSession.builder \
    .appName("YourApp") \
    .getOrCreate()
```

2. **Pass Spark to extractor**:

```
# Before
extractor = PDFSectionExtractor()

# After
extractor = PDFSectionExtractor(spark=spark)
```

3. **Work with DataFrame instead of list**:

```
# Before
sections = extractor.extract_from_document(...)
for section in sections:
    print(section)

# After
sections_df = extractor.extract_from_document(...)
sections_df.select("value").show()

# Or convert to list if needed
sections_list = [row.value for row in sections_df.collect()]
```

1.7.2 Helper Methods

The `get_section_by_keyword()` and `extract_metadata()` methods still work with lists. To use them:

```
# Extract DataFrame
sections_df = extractor.extract_from_document(...)

# Convert to list for legacy methods
sections_list = [row.value for row in sections_df.collect()]
```

```python
# Use helper methods
metadata = extractor.extract_metadata(sections_list)
matching = extractor.get_section_by_keyword(sections_list, 'keyword')
```

Note: These helper methods may be deprecated in future versions in favor of DataFrame operations.

1.8 Performance Benefits

1.8.1 1. Lazy Evaluation

DataFrames use lazy evaluation, so operations are optimized:

```python
# No computation until show() or collect()
filtered = sections_df.filter(sections_df.page_number > 5)
result = filtered.select("value")
result.show()  # Only now does Spark execute
```

1.8.2 2. Distributed Processing

For large-scale processing:

```python
# Process 1000s of documents in parallel
all_docs_df = extractor.extract_from_multiple_documents(
    database='skol_dev',
    doc_ids=large_list_of_ids  # Processes in parallel
)
```

1.8.3 3. Columnar Storage

Write to Parquet for efficient storage and fast queries:

```python
sections_df.write.parquet("sections.parquet")

# Later, fast filtering without reading all data
df = spark.read.parquet("sections.parquet")
df.filter(df.page_number == 1).show()
```

1.9 Integration with SKOL Classifier

The DataFrame output integrates seamlessly with SKOL classifier:

```python
from pdf_section_extractor import PDFSectionExtractor
from skol_classifier.classifier_v2 import SkolClassifierV2

# Extract sections as DataFrame
```

```
extractor = PDFSectionExtractor(spark=spark)
sections_df = extractor.extract_from_document(
    database='skol_dev',
    doc_id='document-id'
)

# Use with SKOL classifier
# Option 1: Convert to list for annotation
sections_list = [row.value for row in sections_df.collect()]
# ... annotate sections ...

# Option 2: Write to temp file
sections_df.select("value").write.text("temp_sections.txt")
# ... use for training ...
```

1.10 Requirements

- **PySpark**: Required for DataFrame output

  ```
  pip install pyspark
  ```

- **SparkSession**: Must be initialized and passed to extractor

1.11 See Also

- PDF_EXTRACTION.md - Original PDF extraction documentation
- example_pdf_extraction.py - Complete examples with DataFrames
- pdf_section_extractor.py - Implementation

Update Date: 2025-12-22 **Status**: ✅ Complete and tested **Breaking Change**: Yes - requires PySpark and Spark initialization **Benefits**: Rich metadata, powerful querying, better scalability