ChatGPT

Targeted Patches for Remaining Limitations in pdf-docx-to-json-docling-v1

The following patches address the known remaining edge cases and limitations in the **pdf-docx-to-json-docling-v1** repository. Each patch is form-agnostic (no hardcoded form specifics) and focuses on one issue. For each patch, we list the relevant file and function, a description of the change, its purpose, and a code snippet illustrating the fix.

Patch 1: OCR Fallback for Partially Scanned PDFs

File: `docling_extract.py` – **Function**: `extract_text_from_pdf`
Description: Modify the PDF text extraction logic to detect and handle **partially scanned PDFs**. Currently, the code auto-detects completely scanned PDFs (no text layer) and triggers OCR for the whole document [1]. This patch adds page-level checks so that if some pages lack text, those pages are processed with OCR while others use the normal text layer.
Purpose: Ensure no content is lost when a PDF has *mixed* pages (some with text, some as images) [2]. This improves robustness by automatically performing OCR on pages with no extractable text, without requiring the user to manually intervene.

```
def extract_text_from_pdf(file_path: Path, use_ocr: bool = False, force_ocr:
bool = False, auto_ocr: bool = True) -> str:
    doc = fitz.open(file_path)
    if force_ocr:
        # (existing force OCR logic) ...
    elif not has_text_layer(doc):
        # (existing auto-OCR for fully scanned docs) ...
    else:
        # Normal extraction, with partial OCR fallback for blank pages
        text_parts: List[str] = []
        for page_index in range(len(doc)):
            page = doc[page_index]
            page_text = page.get_text("text").strip()
            if auto_ocr and OCR_AVAILABLE and page_text == "":
                print(f"  [AUTO-OCR] Page
{page_index+1} has no text, performing OCR")
                # Render page to image and run OCR on it (simplified
pseudocode):
                pix = page.get_pixmap(matrix=fitz.Matrix(300/72, 300/72))
                img = Image.frombytes("RGB", (pix.width, pix.height),
pix.samples)
                ocr_text = pytesseract.image_to_string(img)
```

```
                text_parts.append(ocr_text if ocr_text.strip() else "")
            else:
                text_parts.append(page_text)
        text = "\n".join(text_parts)
    doc.close()
    return text
```

This patch ensures that if a PDF has even one scanned page without text, that page's content will be captured via OCR. It addresses the edge case of **mixed-content PDFs** where previously pages without text might have been silently skipped [2].

Patch 2: Enhanced Multi-Field Line Splitting (Sub-fields like "Day/ Evening" Phones)

File: `docling_text_to_modento/core.py` – **Function**: `detect_multi_field_line`
Description: Expand the detection of multiple sub-fields on one line (one label with several blanks) [3]. The current implementation recognizes common sub-field keywords (e.g., Mobile, Home, Work) but misses time-of-day qualifiers like "Day" and "Evening." We add these to the keyword list and adjust parsing for slash-separated labels.
Purpose: Correctly split fields such as **"Phone: Day _ Evening ___"** into two separate fields (`phone_day` and `phone_evening`) instead of one combined field [4]. This improves accuracy for lines that contain multiple pieces of information under a single label.

```
def detect_multi_field_line(line: str) -> Optional[List[Tuple[str, str]]]:
    # ... existing code ...
    sub_field_keywords = {
        'mobile': 'mobile',
        'cell': 'mobile',
        'home': 'home',
        'work': 'work',
        'office': 'work',
        'primary': 'primary',
        'secondary': 'secondary',
        'personal': 'personal',
        'business': 'business',
        'other': 'other',
        'fax': 'fax',
        'preferred': 'preferred',
        # Added time-of-day keywords for phone numbers:
        'day': 'day',
        'evening': 'evening',
        'night': 'night',
    }
    # Normalize any slashes between sub-field labels to spaces (e.g. "Day/
Evening"):
```

```
    remainder = label_match.group(2)
    remainder = re.sub(r'\/', ' ', remainder)
    parts = re.split(r'[_\s]{2,}', remainder)
    # ... continue as existing, checking parts against sub_field_keywords ...
```

With this patch, the parser will identify **time-based sub-fields** (Day, Evening, etc.) as distinct entries [4]. For example, `"Phone: Day _____ Evening _____"` will yield two fields with keys `"phone_day"` and `"phone_evening"` (titles "Phone (Day)" and "Phone (Evening)"), instead of being lumped together. The logic remains generic, using a configurable keyword list rather than hardcoding any specific form's fields.

Patch 3: Include Column Headers in Multi-Column Checkbox Grids

File: `docling_text_to_modento/modules/grid_parser.py` – **Function**: `parse_multicolumn_checkbox_grid`
Description: Utilize detected **category header labels** in multi-column checkbox grids to group options [5]. The detection function already identifies a header line (e.g., `"Appearance / Function / Habits"`) and captures the category names [6]. This patch prefixes each checkbox option with its column's header during parsing.
Purpose: Preserve the semantic grouping of multi-column checklist options by column. Currently, all options are output as one flat list, losing the column grouping context [5]. With this change, if a grid has labeled columns, the options will reflect those labels (improving data clarity without altering the underlying form-agnostic JSON structure).

```
def parse_multicolumn_checkbox_grid(lines: List[str], grid_info: dict, debug:
bool=False) -> Optional[Question]:
    # ... after collecting item_text for each checkbox ...
    if category_headers and len(category_headers) == len(column_positions):
        # Determine which column this checkbox is in based on its X position
        col_idx = next((idx for idx, col_pos in enumerate(column_positions) if
cb_pos >= col_pos), 0)
        if col_idx < len(category_headers) and category_headers[col_idx]:
            item_text = f"{category_headers[col_idx]} - {item_text}"
    all_options.append((item_text, None))
    # ... continue with de-duplication and Question creation ...
```

Now, given a grid with column headers (say columns **"Appearance", "Function", "Habits"** and options under each), an option like "Smoking" in the third column will be output as **"Habits – Smoking"** [7] [8]. This way, the JSON retains the category context. The approach is the chosen **Option A (prefix)**, which keeps the solution simple and backward-compatible [9]. It does not hardcode any specific headers, instead using whatever text is present in the form's header line.

Patch 4: Improved Inline Checkbox Recognition

File: `docling_text_to_modento/core.py` – **Section**: Inline checkbox detection in `parse_to_questions`

Description: Detect and isolate checkboxes that appear **mid-sentence or mid-paragraph** so they become their own boolean fields [10]. The current regex patterns capture checkboxes at line starts or in lists, but an inline checkbox (e.g., "I agree [ ] to the terms…") may be overlooked. This patch adds a check for any single unchecked box token (`[ ]`) that is **not at the start of the line**, and splits the line around it. The text before and after the checkbox are combined to form a meaningful field label.
Purpose: Ensure no opt-in/opt-out checkbox is missed. This handles cases where the form has a checkbox embedded in a sentence (for example, "[ ] Yes, send me text alerts" or "I agree [ ] to the terms…") so that each such checkbox generates a distinct JSON field [10]. It prevents these inline options from being treated as mere text or merged with another field.

```
for i, raw in enumerate(lines):
    line = raw.strip()
    # ... existing parsing logic ...

    # Detect a lone checkbox mid-line that isn't already handled as an option
list
    if line.count("[ ]") == 1 and not line.lstrip().startswith("["):
        match = re.match(r'^(.*)\[\s*\]\s*(.+)$', line)
        if match:
            prefix_text = match.group(1).strip()
            checkbox_label = match.group(2).strip()
            # Combine prefix and suffix if prefix provides context
            combined_label = f"{prefix_text} {checkbox_label}".strip() if
prefix_text else checkbox_label
            field_key = slugify(combined_label)
            # Create a boolean field (single checkbox question)
            questions.append(Question(field_key, combined_label, cur_section,
"checkbox"))
            if debug:
                print(f"  [debug] Inline checkbox found -> '{combined_label}'")
            continue  # skip further processing of this line
    # ... continue other parsing ...
```

With this addition, a line like:
`"Contact preference: [ ] Yes, email is okay"`
will produce a field **"Yes, email is okay"** (key `yes_email_is_okay`) as a checkbox tied to the context "Contact preference." Similarly, a standalone sentence "I agree [ ] to the terms and conditions" becomes a boolean field **"I agree to the terms and conditions"**. This covers inline checkbox patterns that were previously missed, without misidentifying regular list checkboxes (we only trigger when exactly one checkbox is present in the line).

Patch 5: Flag Potential Visual-Layout-Only Fields

File: `docling_text_to_modento/core.py` – **Function**: `parse_to_questions` (main loop)
Description: Add a diagnostic check to catch lines that likely represent **fields separated only by spacing**, which the text-based parser can miss [11]. If a line contains what looks like multiple input blanks or values

with no label (e.g., two or more sequences of underscores or blanks separated by large spaces) and wasn't parsed by earlier logic, log a warning or debug message.
Purpose: Purely text-based parsing struggles with fields distinguished only by spatial layout [11]. While a full solution would require layout analysis, this patch *at least identifies* such cases so they're visible in debug output (or could be post-processed). It helps developers or users notice where a field might have been skipped due to lack of textual cues.

```
# (At the end of processing each line in parse_to_questions)
clean_line = line.replace("\u00A0", " ")  # replace non-breaking spaces if any
if re.match(r'^\s*_{3,}\s+_{3,}', clean_line) or re.match(r'^\s*(___|…)\s+
(___|…)', clean_line):
    # Line has multiple blank underscores (or similar placeholders) with no
label
    print(f"[WARN] Line {i+1}: Detected multiple blank fields with no label –
parser may skip these.")
```

For example, if the raw text line was just:
`"_____ _____"`
the parser alone won't create two fields. With the patch, it would emit a warning indicating **unlabeled multiple blanks** were found, alerting that a manual check or a future layout-based enhancement is needed [11]. This change primarily **documents the limitation in runtime logs** and does not alter JSON output, preserving current behavior while highlighting the limitation.

Patch 6: Test Coverage Scaffolding for Edge Cases

File: *New test file*, e.g. `tests/test_edge_cases.py`
Description: Introduce a basic **unit test suite** structure to cover the above scenarios [12]. This patch creates a test file and adds sample tests for multi-field splitting, inline checkbox detection, etc. (using pytest). It uses small input strings to verify the functions produce the expected parsed output.
Purpose: Establish a foundation for automated testing so that future changes do not re-introduce these edge-case bugs [13]. By codifying the expected behavior (e.g., that "Day/Evening" splits into two fields, that inline "[ ] Yes…" yields a boolean field), we ensure the fixes remain effective and make it easier to expand test coverage going forward [12].

```
import pytest
from docling_text_to_modento import core

def test_multi_field_label_splitting():
    line = "Phone: Day _____ Evening _____"
    fields = core.detect_multi_field_line(line)
    # Expect two sub-fields: phone_day and phone_evening
    assert fields == [("phone_day", "Phone (Day)"), ("phone_evening", "Phone
(Evening)")]

def test_inline_checkbox_extraction():
```

```
    text_line = "I agree [ ] to the terms and conditions"
    questions = core.parse_to_questions(text_line)
    # The line should produce one Question with title containing the full
sentence without [ ]
    titles = [q.title for q in questions]
    assert "I agree to the terms and conditions" in titles
```

The above tests (as examples) validate the new behavior for multi-subfield lines and inline checkboxes. This patch would also include adding a testing framework (e.g. ensuring **pytest** is installed and a basic `tests/` directory exists) and can be expanded with more cases (checkbox grids, etc.). The goal is to reach at least a minimal coverage for critical parsing functions [14], preventing regressions on these edge cases.

Patch 7: Final Modularization – Separate Parsing Logic into Modules

File: `docling_text_to_modento/core.py` (and new module files)
Description: Finish refactoring the monolithic parser into a modular structure [15]. The core parsing loop (`parse_to_questions` and related helpers) should be moved into the dedicated module (e.g. `modules/question_parser.py`), and other large functions split into `modules/grid_parser.py`, `modules/postprocessing.py`, etc., as planned. This patch creates the `question_parser` module, moves the code, and updates imports accordingly.
Purpose: Improve maintainability and clarity by organizing code into logical components [15]. The parsing script was initially ~5000 lines in one file; this final step reduces that by distributing code into the module structure (which has already been scaffolded). It makes it easier to navigate and modify specific parts (text preprocessing, question parsing, grid handling, etc.) without affecting unrelated parts.

```
# **In docling_text_to_modento/core.py** – replace inline parse_to_questions
with import
from .modules.question_parser import parse_to_questions
# use the function from the module

# ... (remove the old parse_to_questions function implementation) ...

# **In docling_text_to_modento/modules/question_parser.py** – new module for
question parsing
from ..modules import grid_parser, text_preprocessing, ...  # import needed
utilities and modules

def parse_to_questions(text: str, debug: bool=False) -> List['Question']:
    """
    Parse raw text lines into Question objects (moved from core.py).
    """
    lines = [text_preprocessing.normalize_glyphs_line(x) for x in
text_preprocessing.scrub_headers_footers(text)]
    lines = text_preprocessing.coalesce_soft_wraps(lines)
    # ... (rest of parsing logic, iterating over lines and using helper
```

```
functions) ...
    return questions
```

After this patch, `core.py` mainly orchestrates calls into modules rather than containing all logic. Functions like `parse_to_questions`, special detectors, and grid parsing routines reside in their respective modules (as indicated by the existing package structure) [16]. This completes the refactoring marked "**IN PROGRESS**" [15], resulting in cleaner separation of concerns. All existing functionality remains the same – the change is purely organizational, so it should not affect output or parsing behavior.

Each of the above patches can be applied independently to address its specific issue. Together, they bring the project in line with the desired improvements and ensure the pipeline handles edge cases like partial scans, multi-field lines, multi-column groups, and inline checkboxes, all while keeping the solution general-purpose and maintainable. The patches also lay groundwork (testing and modularization) for easier future enhancements.

Sources: The proposed changes are informed by the project's own roadmap and TODOs, including the Actionable Items document and code comments for priorities 1.1, 2.1–2.4, 3.2, and 3.1 [1] [4] [5] [10] [11] [12] [15], which highlight these exact areas as needing improvement. The code snippets provided illustrate how to implement the solutions in a form-agnostic way, without hardcoding for any specific form.

---

[1] [2] [3] [4] [5] [7] [9] [10] [11] [12] [13] [14] [15] [16] ACTIONABLE_ITEMS.md
https://github.com/rontavious999/pdf-docx-to-json-docling-v1/blob/d783747a0cee24353132ebd7e31579cc06613a17/ACTIONABLE_ITEMS.md

[6] [8] grid_parser.py
https://github.com/rontavious999/pdf-docx-to-json-docling-v1/blob/d783747a0cee24353132ebd7e31579cc06613a17/docling_text_to_modento/modules/grid_parser.py