Project Report: PDF OCR and Synthetic Data Generator

Author: Eren Alpar
Github: https://github.com/enadream/OCR-Streamlit
Date: June 15, 2025
Version: 1.0

Executive Summary

This report provides a technical overview of the **PDF OCR Extraction and Synthetic Data Generator**, an application I developed in Python with a Streamlit user interface. The primary requirement was to build a system capable of extracting content (text, images) from PDF documents, with special consideration for the challenges posed by low-quality or scanned source files. The resulting application is a dual-component system. The first component is the **OCR Extractor**, a processing pipeline I designed to convert PDF pages to images, perform skew correction, analyze page layout, extract text, and apply an AI-based spell correction model. The second component is a **Synthetic PDF Generator**, which I designed to programmatically create test documents with simulated real-world scanning artifacts. I delivered the application through a web-based UI for file handling, parameter configuration, and results visualization.

1. Introduction

The main objective of this project was to build an effective OCR engine. When working with Optical Character Recognition (OCR), I found that the accuracy of the results is directly impacted by the quality of the source document. Common issues like page skew, sensor noise, or poor contrast are frequent in scanned archives and can significantly degrade performance.

To address these challenges, my approach involved two key components that work together:

1. **An Advanced OCR Engine:** Rather than a simple OCR tool, I built a full pipeline. This process starts by improving the source image quality, then segments the page to separate text from images, and finally uses a language model to correct OCR errors. I chose this pipeline structure to systematically handle potential issues at each stage of the process.
2. **A Synthetic PDF Generator:** To ensure the OCR engine could handle a variety of real-world conditions, I built a tool to simulate them. This generator programmatically adds flaws like blur and skew into clean PDFs, which provided me with a reliable way to test and refine the engine's performance.

This report details the architecture and technical implementation of this two-part system I built.

2. Objectives

The main technical objectives I set for this project were:

- **Build a Web Application:** Use Streamlit to create an interface for file upload, parameter selection, and results display.

- **Create an OCR Pipeline:** Implement a sequence of steps—image conversion, preprocessing, layout analysis, and text extraction—that work in concert to improve accuracy.
- **Support Multiple OCR Libraries:** Allow the user to select either Tesseract or EasyOCR as the backend OCR engine.
- **Use AI to Fix Mistakes:** After OCR, use a spaCy language model to automatically check and correct spelling errors.
- **Make Realistic Test Files:** Build a feature to take a clean PDF and add simulated scanner artifacts like blur, skew, noise, and ink smudges.
- **Provide Structured Output:** Ensure the final extracted data is delivered in a JSON format that details the content, type (text or image), and location of each detected element on the page.

3. System Architecture and Design

The application is designed with a modular, pipeline-based architecture.

- **Code Organization:**
    - `app/core`: Contains the backend logic for the OCR pipeline, with modules for PDF handling, image processing, layout detection, and extraction.
    - `app/utils`: Contains helper modules, including the `synthetic_generator` and the `spell_checker`.
    - `app/ui`: Contains all Streamlit code for the user interface.
- **The OCR Extractor Pipeline in Detail:** The extraction process is a chain of operations where the output of one step is the input for the next.
    - **Pipeline Step 1: PDF to Image Conversion (`pdf_handler.py`)** The process begins when a user uploads a PDF. The selected pages are converted into high-resolution PNG images.
    - **Pipeline Step 2: Image Preprocessing (`image_processor.py`)** Each image is analyzed for rotational skew. If a page is crooked, it is algorithmically straightened. This step is critical for the subsequent layout analysis.
    - **Pipeline Step 3: Layout Analysis (`layout_detector.py`)** Using the straightened image, the code identifies content regions, drawing bounding boxes around text paragraphs and images. Each box is assigned a unique ID (e.g., `text_1`, `image_1`).
    - **Pipeline Step 4: OCR and Content Extraction (`ocr_extractor.py`)** The system processes the regions identified in the previous step. Text regions are passed to the selected OCR engine. Image regions are cropped and saved as separate image files.

    - **Pipeline Step 5: AI-Powered Spell Correction (`spell_checker.py`)** The raw text output from the OCR engine is processed by a spaCy language model, which identifies and corrects contextual spelling errors.
    - **Pipeline Step 6: Final Output Generation (`ocr_extractor.py`)** All corrected text, paths to saved images, and bounding box coordinates are aggregated and saved into a single JSON file.
- **The Synthetic Generator Workflow:** This process focuses on adding artifacts to clean documents.
    - **PDF to Image:** A clean PDF is converted into a series of images.
    - **Add Artifacts:** A stack of image effects (blur, skew, noise, etc.) is randomly applied to each image based on probabilities set by the user in the UI.
    - **Images to PDF:** The augmented images are reassembled into a single PDF that simulates a document with scanning defects.

4. Technical Implementation Details

This section provides a deeper look into the specific algorithms, image processing techniques, and configuration parameters used in the project.

Multi-Language Support

I designed the application to handle multiple languages, including English and Turkish, by integrating the language-specific models of the OCR engines and spell-checking libraries.

- **OCR Engines:**
    - **Tesseract:** The `ocr_extractor.py` module maps user-selected languages (e.g., 'English', 'Turkish') to the corresponding 3-letter ISO codes required by Tesseract ('eng', 'tur'). These language models must be installed on the system.
    - **EasyOCR:** This library natively accepts 2-letter language codes ('en', 'tr'). My code includes a caching mechanism (`get_easyocr_reader`) so that once a language model is loaded for the first time, it is kept in memory for subsequent requests, which improves performance.
- **Spell Correction:** The `spell_checker.py` module uses a dictionary to map language codes to the correct spaCy model name (e.g., `'en': 'en_core_web_sm'`, `'tr': 'tr_core_news_sm'`). Similar to EasyOCR, I implemented a cache so that spaCy models are only loaded once per session.

Image Preprocessing: Skew Correction

The `correct_skew` function in `image_processor.py` is used to improve OCR accuracy. It operates using a **projection profile method**:

1. **Binarization:** The input image is converted to grayscale and then binarized using `cv2.THRESH_OTSU`, which automatically determines the threshold value. The image is inverted so that text pixels are white (foreground) and the background is black.
2. **Angle Search:** The algorithm iterates through a predefined range of angles (e.g., -15° to +15° in 0.5° increments).
3. **Rotation and Scoring:** For each angle, the binary image is rotated. A "sharpness" score is then calculated for the rotated image based on its **horizontal projection profile**.
4. **Final Rotation:** The angle that yields the highest score is identified as the skew angle. The original, full-color image is then rotated by this angle.

Layout Analysis: Content Segmentation

The `detect_content_regions` function in `layout_detector.py` uses a multi-step, heuristics-based approach with OpenCV:

1. **Image Region Detection:** To prevent large images from being misidentified as text, the algorithm first finds all external contours and classifies any contour exceeding a configured area threshold (`IMAGE_AREA_THRESHOLD_PERCENT`) as an "image". These regions are then masked out.
2. **Text Block Detection:** The algorithm then uses morphological operations (dilation with horizontal and vertical kernels) on the remaining parts of the image to merge individual words and lines into cohesive text blocks.
3. **Contour Extraction:** Contours are drawn around these merged text blocks. Any contour larger than `MIN_CONTOUR_AREA` is classified as a "text" region.
4. **ID Assignment and Sorting:** All detected regions are assigned a sequential ID and sorted vertically to ensure a logical reading order.

Synthetic Data Generation: Augmentation Techniques

The `ScannedDocumentAugmentor` class applies a series of randomized OpenCV-based augmentations: Skew, Gaussian Noise, Gaussian Blur, Brightness/Contrast adjustments, and Ink Smudges. Each effect's application is controlled by a probability slider in the UI.

Configuration Management

The application's behavior is controlled by parameters in `config.py` files.

- **`app/core/config.py`:** Controls file paths, image DPI, and key parameters for layout detection (`MIN_CONTOUR_AREA`, `IMAGE_AREA_THRESHOLD_PERCENT`, `TEXT_DILATION_KERNEL_X/Y`). It also contains boolean flags for saving intermediate debug images.
- **`app/utils/synthetic_generator/config.py`:** Manages paths and settings specific to the data generation workflow.

5. Installation Guide

A detailed installation guide with step-by-step instructions for Linux, Windows, and macOS can be found in the `README.md` file. This file is available in the project's root directory and on the main page of the GitHub repository.

6. How to Use the Application

After installation, run the app from the project's root directory:

```
python -m app.main
```

The application has two modes, selectable from the sidebar.

Using the PDF OCR Extractor

1. Select "PDF OCR Extractor" from the sidebar.
2. Upload a PDF file using the file uploader.
3. In the "OCR Settings" sidebar, choose the document language (e.g., English, Turkish).
4. Select the OCR engine (Tesseract or EasyOCR).
5. Optionally, enable or disable the AI spell correction.
6. Specify which pages to process (e.g., `all`, `1, 5`, `2-8`).
7. Click the "Process Document" button.
8. The application will display its progress through the pipeline steps.
9. Once complete, the results will appear. You can view a labeled debug image showing the detected content regions and expand each region to see the extracted text or image.

Using the Synthetic PDF Generator

1. Select "Synthetic PDF Generator" from the sidebar.
2. Upload a clean PDF file.
3. In the sidebar, use the sliders to adjust the probability for each type of artifact (Blur, Skew, Noise, etc.).
4. Click the "Generate Synthetic PDF" button.
5. When the process is finished, a download button will appear, allowing you to save the newly generated PDF.

7. File Structure

The project is organized into the following directory structure:

```
project/
|---- app/
|    |---- __init__.py
|    |---- main.py
|    |---- core/
|    |    |---- __init__.py
|    |    |---- config.py
|    |    |---- image_processor.py
|    |    |---- layout_detector.py
|    |    |---- ocr_extractor.py
|    |    |---- pdf_handler.py
|    |---- data/
|    |    |---- (This directory is created dynamically for storing I/O files)
|    |---- ui/
|    |    |---- __init__.py
|    |    |---- main_ui.py
|    |---- utils/
|    |    |---- __init__.py
|    |    |---- spell_checker.py
|    |    |---- synthetic_generator/
|    |    |    |---- __init__.py
|    |    |    |---- config.py
|    |    |    |---- image_augmentor.py
|    |    |    |---- pdf_processor.py
|---- requirements.txt
|---- setup.py
|---- README.md
```

8. Conclusion

The **PDF OCR Extraction and Synthetic Data Generator** meets the project objectives. It provides a tool for OCR extraction using a pipeline that incorporates preprocessing and a correction layer, designed to produce more accurate results than basic OCR methods. The Synthetic PDF Generator component provides a useful capability for creating varied test data. The project effectively addresses common challenges in document digitization and serves as a functional application for document processing and data extraction tasks.