ChatGPT

Multilingual PDF Heading Detection: Architecture and Solution

Automated **heading detection** in PDFs is critical for document understanding, accessibility and downstream tasks. A robust solution must identify section titles with high precision/recall (score focus [1]) while running efficiently in a containerized environment and handling multiple languages (e.g. Japanese). We propose a pipeline that ingests PDFs, extracts layout and text features, classifies heading lines, and outputs a structured outline. This modular architecture (parse → features → classify → output) follows industry best practices. For example, enterprise workflows separate "ingestion" and "retrieval" stages for PDF content [2] (see Figure 1). We adapt this idea: our *ingestion* parses and OCRs pages, and our *analysis* stage performs heading classification. Importantly, our solution is **self-contained** (no external web calls) and **lightweight**, aligning with competition constraints and Adobe's focus on efficient on-device document AI.

Figure: Example architecture of a multi-stage PDF processing pipeline. We parse PDFs into text runs (using PyMuPDF or PDFMiner) and, if needed, fall back to OCR (Tesseract/EasyOCR). Each text block's style (font size, weight, layout) and content (word count, POS tags, numerals, etc.) are extracted as features. For instance, features like "font size", "bold flag", "word count", and linguistic features (text case, number of nouns/verbs) have proven highly predictive of headings [3] [4]. A lightweight classifier (e.g. decision tree or logistic regression) is trained on labeled examples to tag each block as *heading* or *body*. Detected headings are then organized into a hierarchy (levels inferred by relative font sizes or numbering). This process yields a structured output (JSON or tagged PDF) representing the document's outline.

System Architecture & Pipeline

Our system consists of modular components, each containerized via Docker for portability:

- **PDF Ingestion:** Read the input PDF using a library like PyMuPDF or pdfplumber. These tools extract text spans along with styling (font name, size, bold/italic). If the PDF has *no extractable text* (scanned pages), we convert pages to images and apply OCR. We integrate open-source OCR engines: Tesseract (supports >100 languages [5]) and EasyOCR (supports 80+ languages including Japanese [6]). These engines produce text with position info and preserve style cues (e.g. bounding boxes, pixel intensity for bold).
- **Feature Extraction:** For each text block or line, compute features capturing visual and textual cues. Key features include:
- **Typography:** font size relative to document, bold/italic flags, font color or hue, indentation/margin. Headings often use larger, bold fonts [3]. (Prior work confirms font/style features boost heading detection [4].)
- **Layout:** Y-coordinate or distance from top (e.g. headings often at top of page/section), whitespace above/below.

- **Textual:** word count, presence of numbering or punctuation (e.g. "1." or "-"), capitalization patterns, POS tag counts (verbs, nouns, numbers) [3]. For example, headings often contain few words and specific content (like "Introduction"), whereas body text has more verbs/nouns.
- **Heading Classification:** Apply a machine learning model (trained beforehand on labeled PDF data) to classify each block. We favor *lightweight supervised models* (e.g. decision trees, logistic regression, or a small neural net) to meet performance constraints. Recursive feature elimination on similar problems found that ~7–9 features suffice [3], enabling compact models. We can also include simple rule-based overrides (e.g. treat text in largest font as heading) without hardcoding specific words. The classifier outputs a label (heading or not) and optionally a level.
- **Post-processing:** Assemble detected headings in document order, inferring hierarchy. For instance, if heading A has font size 16pt and heading B 12pt on the same page, B might be a subheading under A. We generate a structured Table-of-Contents (TOC) or annotate the PDF with heading tags. This structured output greatly aids navigation and accessibility (see below).

Each component runs as a microservice or module, allowing parallel processing (e.g. parsing pages concurrently) for speed. The entire pipeline runs offline in Docker, so no network calls are required, satisfying the "no API/web calls" rule.

Key Features

- **High Accuracy Heading Detection:** By combining visual style and text features in a trained classifier, we target precision and recall above 90%. Prior work on similar tasks achieved ~97% accuracy using such features [1]. Our model is tuned on a diverse PDF corpus to generalize to different layouts.
- **Lightweight Models & Speed:** We use compact ML models and efficient libraries. For example, EasyOCR is *designed* to be lightweight (works on CPU with no GPU needed [7]), and Tesseract is highly optimized C++. Models (or coefficients) are small (under a few MB) to meet size constraints. The pipeline processes pages quickly; on modern hardware we expect processing times on the order of 0.1–0.5 seconds per page.
- **Containerized Deployment:** The solution is fully Dockerized (Dockerfile at repo root). All dependencies (Python, OCR engines, libraries) are installed inside the container. This ensures consistent environments and aligns with the submission checklist. We include entrypoint scripts or a simple CLI (e.g. `detect_headings input.pdf output.json`) for ease of use.
- **Modularity:** Code is organized into reusable modules (e.g. `PdfParser`, `OcrProcessor`, `FeatureExtractor`, `HeadingClassifier`). This modular design enables future extension (e.g. adding a deep-learning model in Round 1B) and makes testing easier. Each module can be developed and tested independently.
- **Multilingual Support:** The pipeline is language-agnostic for the most part. Since our classifier relies on layout and simple text features, it does not depend on English-specific tokens. When OCR is needed, we load appropriate language models (e.g. Japanese) for Tesseract/EasyOCR. Both OCR engines support Japanese and many languages [5] [6]. This ensures headings in Japanese, Chinese, or other languages are recognized and processed.
- **Accessibility & Social Impact:** Automatic heading detection improves document accessibility. Properly tagged headings allow screen readers and assistive technology to navigate content easily [8] [9]. By making PDFs structured (e.g. generating `<H1>, <H2>, …` tags), our solution supports the creation of accessible content. This aligns with public demand for inclusive tools and Adobe's

commitment to accessibility. Moreover, our open-source approach (using permissive libraries and sharing code after the deadline) maximizes community benefit.
- **Robustness:** We explicitly avoid brittle heuristics or hardcoding. For instance, we do not assume fixed fonts for headings, since some documents may use same font size at different levels. Instead, the model learns patterns. We also ensure broad coverage by testing on varied PDFs (textbooks, reports, articles, etc.) as recommended. Error handling (e.g. skipping empty pages) and logging are included for production readiness.

Technology Stack

- **Python 3** – main programming language for flexibility and rich libraries.
- **PDF Libraries:** PyMuPDF or pdfplumber to extract text with font/style. These provide fast, structured access to PDF content.
- **OCR Engines:** Tesseract OCR via `pytesseract`, and/or EasyOCR. Both support multi-language text. Tesseract can run fully offline on CPU and supports training for fonts, and EasyOCR offers robust detection with minimal setup [10] [6].
- **Machine Learning:** scikit-learn for training and running classifiers (logistic regression, decision trees, etc.). We may also consider XGBoost if tree ensembles give an edge in accuracy, as they still fit the "lightweight" constraint when small.
- **NLP Tools:** For textual features, spaCy or NLTK can provide part-of-speech tagging if needed. However, to keep dependencies minimal, we may use simple heuristics (capitalization, counting vowels/numbers) to estimate word types.
- **Containerization:** Docker – the Dockerfile installs all dependencies. This meets the requirement of a working Dockerfile and reproducible environment.
- **Testing:** pytest for unit tests on each component (parsing, OCR, classification). Automated tests ensure our code passes example PDFs (simple and complex) and checks that expected headings are extracted.
- **Documentation:** A `README.md` explains the approach, model choice, and step-by-step build/run instructions. This complies with submission guidelines (documenting how to build and run using the expected CLI).

Performance & Compliance

To satisfy the **"Time & Size Compliance"** scoring (10 points), we focus on efficiency: minimal dependencies (no heavy DL unless absolutely needed) and optimized code paths (e.g. reuse OCR results, vectorize operations). We will benchmark on sample PDFs to ensure fast average processing. For instance, EasyOCR's CPU mode is very efficient [7], and PyMuPDF can parse text in milliseconds per page. By quantifying performance during development, we ensure the solution runs within any time limits. We also keep model sizes small (under a few MB) to not exceed size constraints. The Docker image will be lean (using a slim Python base image).

We comply with all given **constraints and submission requirements**. The Git repo includes a Dockerfile in the root that installs dependencies (no additional manual setup needed) [11]. All required libraries (PyMuPDF, scikit-learn, pytesseract, etc.) are listed in `requirements.txt` and installed in the container. The `README.md` documents our entire approach (as outlined here) and lists any model/library used (e.g. we note if we use a pre-trained classifier or manually labeled data for training). We will instruct how to build the Docker image and run the solution (e.g. `docker build .` then `docker run -v $(pwd):/data` `heading-detector input.pdf`). The code has no file-specific logic and no hardcoded formats – it generalizes to any PDF.

Roadmap (3-Week Plan)

Our small team of 3 will work in parallel sprints, with clear deliverables each week:

1. **Week 1 – Setup & Basic Parsing:** Set up project and Docker environment. Research and gather representative PDFs (varied layouts, languages). Implement PDF parsing module: use PyMuPDF/pdfplumber to extract text spans and style. Build simple text cleaning. Prototype feature extractor that prints font sizes and content for each block. Set up OCR fallback with Tesseract/EasyOCR and test on a few scanned pages. Write unit tests for parsing. Document data formats.

2. **Week 2 – Model Development:** Label a small training set (using available tools or manual tags) of PDF pages to mark headings. Based on extracted features, train a classifier (start with logistic regression). Tune it to balance precision/recall on a validation set. Incorporate style-based features as per [22] and [25]. Integrate the model into the pipeline, so that running on a PDF outputs candidate headings. Begin measuring accuracy (target F1 > 90%). Write integration tests (given a sample PDF, check extracted headings against ground truth).

3. **Week 3 – Optimization & Polish:** Focus on performance and edge cases. Profile the pipeline; if slow, optimize (e.g. limit OCR only to images with no text, use multiprocessing). Ensure multi-language support: test with Japanese PDFs to verify OCR and classification work. Refine the classifier if needed (e.g. add or remove features). Finalize output format (e.g. JSON TOC or annotated PDF). Conduct thorough testing on both simple and complex PDFs (as advised). Prepare final documentation: complete the README with usage and examples. Ensure Docker image builds cleanly and container runs end-to-end.

At the end of this roadmap, we deliver a complete system meeting all the criteria: high heading detection accuracy, fast execution, multilingual support, and full documentation. This plan is realistic given our team size and timeline, with room for iteration and improvement.

Social Impact and Alignment with Adobe

Our solution has strong social value: by automatically structuring PDFs, it **enhances accessibility** and knowledge sharing. Well-tagged documents allow visually impaired users to navigate content via headings [8] [9], aligning with public demand for inclusivity. Also, researchers and students benefit when large document collections become searchable and summarizable (e.g. easier table-of-contents generation and content summarization).

For Adobe, which champions creativity and digital literacy, our pipeline can integrate into Acrobat or Document Cloud as a plugin/service, providing users immediate structure extraction. Using open-source components shows a commitment to community collaboration, and the containerized architecture fits Adobe's enterprise solutions. In summary, our industry-grade design not only meets technical requirements, but also supports broader Adobe goals and social impact through accessible, structured content.

Sources: Academic and industry research confirm the viability of our approach. For instance, supervised classifiers on font/style features reached ~97% accuracy on heading detection [1], and style-based features are known to improve results [4]. Tools like EasyOCR and Tesseract support the necessary languages [5] [6], and modern cloud services (e.g. Microsoft's Document Intelligence) now include dedicated heading detection [12], underscoring the importance of this task. With these proven methods and a solid implementation plan, our solution is both practical and impactful.

---

[1] [3] [1809.01477] A Supervised Learning Approach For Heading Detection
https://ar5iv.labs.arxiv.org/html/1809.01477

[2] Build an Enterprise-Scale Multimodal PDF Data Extraction Pipeline with an NVIDIA AI Blueprint | NVIDIA Technical Blog
https://developer.nvidia.com/blog/build-an-enterprise-scale-multimodal-document-retrieval-pipeline-with-nvidia-nim-agent-blueprint/

[4] Information Extraction from Visually Rich Documents with Font Style Embeddings
https://arxiv.org/pdf/2111.04045

[5] [6] [7] [10] Easyocr vs Tesseract (OCR Features Comparison)
https://ironsoftware.com/csharp/ocr/blog/ocr-tools/easyocr-vs-tesseract/

[8] [9] Common PDF Tags and Their Usage | Section508.gov
https://www.section508.gov/create/pdfs/common-tags-and-usage/

[11] GitHub - fabriziosalmi/pdf-ocr: Converts scanned PDF documents to multiple formats using Optical Character Recognition
https://github.com/fabriziosalmi/pdf-ocr

[12] What's new in Document Intelligence - Azure AI services | Microsoft Learn
https://learn.microsoft.com/en-us/azure/ai-services/document-intelligence/whats-new?view=doc-intel-4.0.0