Project Documentation: Advanced PDF Heading Detection System

1A) Heading detector

1. Executive Summary

This project presents a robust, machine learning-powered system for automatically detecting and classifying hierarchical headings (e.g., Title, H1, H2, H3) within PDF documents. The primary challenge with PDFs is their lack of structured semantic information; they are primarily visual-first documents. Our solution addresses this by building a sophisticated pipeline that combines deep PDF metadata extraction, extensive feature engineering, and a unique two-stage classification model.

The system is trained entirely on **synthetically generated data**, bypassing the need for a large, manually-annotated dataset. It is designed for high accuracy and resilience, incorporating fallback mechanisms and rigorous data validation at every step. The final output is a structured JSON file detailing the document's hierarchy, providing valuable data for content analysis, document summarization, and automated indexing.

---

2. System Architecture & Core Approach

The system operates through a sequential pipeline, where each component is responsible for a specific task. This modular design makes the system easy to understand, debug, and extend.

Architectural Flow:

`[PDF File]` → **(1) PDF Processor** → `[Raw Text Elements]` → **(2) Feature Extractor** → `[Feature Matrix $X$]` → **(3) Classification System** → `[Predictions]` → **(4) Output Formatter** → `[Structured JSON]`

Core Principles:

1. **Two-Stage Classification:** Instead of a single, complex multi-class model, we use a two-stage approach:
    - **Binary Classifier:** First, determine if a text element is a heading or not (`heading` vs. `non-heading`). This simplifies the initial problem.

    - **Hierarchical Classifier:** Second, for elements identified as headings, classify their specific level (`title`, `h1`, `h2`, `h3`). This model is trained only on heading data, allowing it to focus on the nuances that differentiate heading levels.
2. **Synthetic Data Generation:** We create a rich, varied training dataset from scratch. This generator creates realistic text elements and, crucially, uses the **same feature extractor** as the main pipeline to generate the training data. This ensures perfect consistency between training and prediction features, preventing feature skew.
3. **Rich Feature Engineering:** The model's intelligence comes from a comprehensive set of features (~43 in total) derived from text content, font properties, and positional information.
4. **Robustness and Fallbacks:** The system is built to handle real-world, messy PDFs. The `PDF Processor` uses a fallback mechanism (`PyMuPDF`) if the primary extractor fails, and the `Classifier` uses rule-based fallbacks if model confidence is low.

---

3. Detailed Component Breakdown

3.1. `main.py`:

This script serves as the main entry point for the application.

- **Purpose:** To handle command-line interface (CLI) arguments, initialize all system components, and manage the overall workflow for single-file processing, batch processing, and model training.
- **Key Logic:**
    - Uses Python's `argparse` to define actions like processing a PDF (`pdf`), training the model (`--train-synthetic`), or running a batch job (`--batch-dir`).
    - The `PDFHeadingDetectionSystem` class acts as a high-level wrapper that instantiates and connects the processor, extractor, classifier, and formatter.

Bash
```bash
# Example Usage:
# Train the model with 15,000 synthetic samples and hyperparameter optimization
python main.py --train-synthetic --samples 15000 --optimize-params --force-retrain

# Process a single PDF and display the results
python main.py my_document.pdf --display
```

3.2. `pdf_processor.py`: Text & Metadata Extraction

- **Purpose:** To extract text elements from a PDF while preserving critical metadata like font size, style (bold/italic), and position.
- **Key Logic:**
    - **Primary Method (`_extract_with_layout_analysis`):** Uses `pdfminer.six` to perform a detailed layout analysis. This is the preferred method as it provides rich character-level font information.
    - **Fallback Method (`_extract_with_pymupdf`):** If `pdfminer.six` yields too few elements (a sign of a complex or problematic PDF), the system automatically switches to `PyMuPDF` (fitz). `PyMuPDF` is generally more robust and faster, making it an excellent fallback.
    - **Post-processing (`_post_process_elements`):** After extraction, elements are cleaned, validated, and re-sorted based on their page and vertical position to ensure a logical document order.

3.3. `feature_extractor.py`:

- **Purpose:** To convert raw text elements into a numerical feature matrix (X) that the machine learning models can understand.
- **Key Feature Categories:**
    - **Text Features:** `char_count`, `word_count`, `is_upper`, `is_title`, capitalization ratio, punctuation count, etc. These features capture the stylistic properties of the text itself.
    - **Numbering Features:** `is_numbered` (e.g., starts with "1.1" or "A.") and `numbering_level`. These are powerful indicators of structured headings.
    - **Font Features:** `font_size`, `is_bold`, `is_italic`, and crucial **relative font features** like `font_size_ratio_max` (the element's font size compared to the document's max font size) and `font_percentile`. These relative metrics are more robust than absolute font sizes, as they adapt to each document's specific styling.
    - **Positional Features:** `page_num`, `position` (overall element index), `relative_position` (position within the page), and flags like `is_early_position`. Headings are often located at the top of a page.
    - **Content Quality Features:** Heuristic-based Part-of-Speech (POS) tagging (`noun_ratio`, `verb_ratio`), counts of common heading keywords ("Introduction", "Conclusion"), and flags for standalone lines.
- **Noise Filtering (`_is_noise_element`):** This component includes a highly sophisticated noise filter that uses regular expressions to remove common non-content elements like page numbers, footers, email addresses, and decorative separators before feature extraction.

3.4. `synthetic_data_generator.py`:

- **Purpose:** To create a high-quality, labeled dataset for training the classifiers without any manual effort.
- **Key Logic:**

    - It uses predefined templates for each heading type and for non-heading text.
    - It randomizes topics, numbers, font sizes, and positional attributes to create a diverse set of examples.
    - **Crucial Design Choice:** After generating raw samples (text, font size, etc.), it feeds them into the **`EnhancedFeatureExtractor`**. The final training dataset consists of the features produced by the extractor and the labels of the samples that survived its noise filtering. This guarantees that the features seen during training are generated in the exact same way as the features seen during prediction.

3.5. `classifiers.py`: The Machine Learning Core

This is the central component for training and prediction.

- **Training Process (`train_with_synthetic_data`):**
    1. **Load/Generate Data:** Loads existing synthetic data or generates a new set.
    2. **Preprocess Features:** Applies `StandardScaler` to normalize features and `SelectKBest` to select the most impactful features, reducing model complexity and improving performance.
    3. **Train Binary Classifier:** Creates binary labels (`heading` vs. `non-heading`). It uses a `RandomForestClassifier` with `GridSearchCV` to find the optimal hyperparameters for distinguishing headings from other text.
    4. **Train Hierarchical Classifier:** Filters the dataset to include only heading samples. It then trains a second `RandomForestClassifier` on this subset to classify the specific heading level (`title`, `h1`, `h2`, `h3`).
    5. **Save Models:** The trained classifiers, the feature scaler, and the feature selector are all saved to disk using `joblib`.
- **Prediction Process (`predict`):**
    1. **Pre-computation Filtering (`_passes_text_quality_check`):** It quickly discards elements that are obviously not headings (e.g., too short, too long) *before* running the models.
    2. **Stage 1 - Binary Prediction:** The binary classifier predicts if each element is a `heading` and provides a confidence score.
    3. **Stage 2 - Hierarchical Prediction:** If an element is classified as a `heading` with confidence above a threshold (`BINARY_THRESHOLD ~ 0.75`), it is passed to the hierarchical classifier.
    4. **Confidence & Fallbacks:**
        - If the hierarchical model is also confident (`hierarchical_confidence >= HIERARCHICAL_THRESHOLD`), the prediction is accepted.
        - If the hierarchical model is *not* confident, the system uses a **rule-based fallback** (`_determine_hierarchy_fallback`) to assign a level based on font size and numbering patterns, but with a confidence penalty. This makes the system more practical.

        - The final confidence is a weighted average of the two models' confidences: `final_confidence` = 0.6timestextbinary_confidence+0.4timestexthierarchical_confidence

3.6. `output_formatter.py`: Polished & Insightful Results

- **Purpose:** To convert the raw list of predictions into a structured, informative JSON output.
- **Key Logic:**
    - It constructs a `document_structure` list containing detailed information for each detected heading.
    - It calculates and adds high-level `document_info`, including a heading distribution (`"h1": 5, "h2": 12`).
    - **Value-Added Analysis:** It computes quality metrics like a `structure_score` and assesses if the document is `is_well_structured`. This provides users with an immediate sense of the document's quality and the reliability of the results.
    - It also provides a pretty-printed console view for quick analysis using the `--display` flag.

3.7. `config.py` and `utils.py`

- **`config.py`:** A centralized configuration file. All important parameters—file paths, model thresholds, feature extraction settings—are defined here. This makes the system easy to tweak and maintain without changing the core logic.
- **`utils.py`:** A collection of helper functions used across the project, such as `clean_text`, `is_numeric_heading`, and `get_heading_level`. This promotes code reuse and keeps the main component files clean.

Of course. Here is the updated **Results & Performance** section incorporating the classification report from your image.

---

4. Results & Performance 📊

The system's performance was evaluated on a held-out test set comprising **15,165 samples**. The model demonstrates excellent performance across all heading types, achieving an overall accuracy of **99%**. The detailed classification report below showcases high precision, recall, and F1-scores for each class, indicating a well-balanced and highly accurate model.

4.1. Model Classification Report

Class	Precision	Recall	F1-Score	Support
h1	0.95	1.00	0.97	3000
h2	0.99	0.94	0.97	3000
h3	1.00	1.00	1.00	3000
non-heading	1.00	1.00	1.00	3165
title	1.00	1.00	1.00	3000
Accuracy			**0.99**	**15165**
Macro Avg	0.99	0.99	0.99	15165
Weighted Avg	0.99	0.99	0.99	15165

The above results can be obtained by running python evaluate_model.py in the code for 1B.

4.2. Qualitative Analysis

When tested on a small set of real-world academic papers and technical manuals, the system demonstrated strong qualitative performance:

- **Strengths:** The system excels at identifying standard, single-column document structures. The combination of font-based features (especially relative size) and numbering-based features (`is_numbered`) allows it to correctly classify the hierarchy in well-formatted documents with high accuracy.

- **Areas for Improvement:** The system is currently less effective on complex, multi-column layouts or in documents with highly unconventional formatting (e.g., artistic PDFs, magazines). Text embedded in images or purely scanned (non-digital) PDFs are partially supported.

1B. Persona-Driven Document Intelligence

1. Executive Summary

This project (Part 1B) builds directly upon the heading detection system from Part 1A to create a **Persona-Driven Document Intelligence** engine. The primary challenge this system addresses is information overload; professionals often need to find specific, relevant information scattered across multiple long and complex documents. Sifting through this content manually is inefficient and time-consuming.

Our solution is an intelligent pipeline that understands a user's **persona** (who they are) and their **job-to-be-done** (what they need to accomplish). By combining the structured output from Part 1A with a powerful semantic search and a hybrid ranking model, the system analyzes a collection of documents and automatically identifies, ranks, and extracts the most relevant sections and subsections. The final output is a concise, actionable summary tailored specifically to the user's needs, dramatically accelerating their workflow.

---

2. System Architecture & Core Approach

The system extends the Part 1A pipeline with a sophisticated analysis layer. It processes a user query and a set of documents to deliver a targeted response.

Architectural Flow:

`[Input Spec: Persona, Job, Docs]` → **(1) Persona Processor** & **Part 1A Heading Detector** → `[Structured Persona]` & `[Document Structures]` → **(2) Embedding Engine** → **(3) Section Ranker** → `[Ranked Sections]` → **(4) Subsection Analyzer** → `[Final JSON Result]`

Core Principles:

1. **Foundation on Part 1A:** The system first uses the `PDFHeadingDetectionSystem` (from Part 1A) to parse every document, transforming unstructured PDFs into a structured list of sections with associated content.
2. **Persona Understanding:** A dedicated `PersonaProcessor` analyzes the natural language descriptions of the user's role and task, converting them into a structured profile with keywords, expertise areas, and a primary domain.
3. **Semantic Search Core:** The `EmbeddingEngine` uses a state-of-the-art sentence-transformer model (`all-mpnet-base-v2`) to convert the persona query and all document sections into numerical vectors (embeddings). This allows the system to match content based on **semantic meaning**, not just keywords.
4. **Hybrid Relevance Ranking:** The `SectionRanker` is the system's brain. It calculates a relevance score for every section using a hybrid, weighted formula. This prevents over-reliance on any single metric and produces a more robust ranking. The formula is:
    - `Relevance = (w_s * Semantic) + (w_d * Domain) + (w_k * Keywords) + (w_t * Type) + (w_q * Quality)`
5. **Progressive Summarization:** The system first identifies the most relevant **sections**. It then performs a deeper analysis on only these top-ranked sections, breaking them down into concise, refined **subsections** and extracting key concepts. This ensures the final output is both relevant and highly digestible.

---

3. Detailed Component Breakdown

- **`main_1b.py`: Main Entry Point**
    - Handles the overall execution, parsing command-line arguments, and managing I/O. It reads an `input_spec.json` file containing the persona, job, and document list, and writes the final `analysis_result.json`.
- **`persona_document_analyzer.py`: The Orchestrator**
    - This is the central class that manages the entire analysis pipeline. It initializes all other components and calls them in the correct sequence: process persona, extract document structures, rank sections, analyze subsections, and build the final output.
- **`persona_processor.py`: Persona Understanding**
    - Uses rule-based logic and keyword extraction to parse the user's `persona` and `job_to_be_done`. It identifies the user's role (e.g., "HR

Professional"), domain ("HR"), expertise areas, and key concepts from their task description to create a comprehensive profile.

- **`embedding_engine.py`: Semantic Core**
    - A critical component that loads and manages the `all-mpnet-base-v2` sentence-transformer model. It's optimized for offline, CPU-only execution to meet platform constraints. Its primary functions are to encode text into 768-dimension vectors and calculate the cosine similarity between them, enabling powerful semantic matching.
- **`section_ranker.py`: Relevance Ranking Engine**
    - This module iterates through every section from every document and scores them against the persona profile. It calculates the hybrid relevance score by combining semantic similarity, domain keyword matches, task keyword matches, section type (e.g., "Introduction" vs. "How to"), and content quality.
- **`subsection_analyzer.py`: Content Refinement**
    - Takes the top-ranked sections and performs a deeper dive. It splits the section's raw content into smaller, logical subsections (e.g., paragraphs or groups of sentences). It then scores these subsections for relevance, refines the text for clarity, and extracts key concepts, providing the final layer of summarization.
- **`config_1b.py`: Centralized Configuration**
    - Contains all key parameters for the system, including the embedding model name, text length limits, and, most importantly, the **weights** for the hybrid scoring model in the `SectionRanker`. This allows for easy tuning of the ranking algorithm.