PROJECT REPORT

Technical Implementation & Testing Manual

for

DocLayout AI

Prepared by:

1. Archit Jaju (IMT2023128)
2. Sanyam Verma (IMT2023040)
3. Pushkar Kulkarni (IMT2023087)
4. Harjot Singh (IMT2023064)

December 7, 2025

Contents

1 Project Overview

DocLayout AI is an offline, command-line system designed to automate the process of identifying and ranking relevant sections from PDF documents based on a specific user persona and job-to-be-done (JTBD).

The system addresses the challenge of information overload by using a **persona-driven approach**. Instead of simple keyword matching, it employs semantic analysis to determine contextual relevance, producing structured insights in a standard JSON format. The project was architected to strictly adhere to constraints regarding offline execution (no external APIs) and efficient CPU-based processing.

2 Technical Implementation Phases

The development was divided into three distinct phases to ensure modularity and performance.

2.1 Phase 1: Structural Parsing (Heuristic Engine)

Objective: Rapidly identify document structure (headings vs. content) without heavy computation.

To meet the requirement of processing 3-5 documents in under 60 seconds on a CPU, we avoided Deep Learning OCR models (like PaddleOCR) which are resource-intensive. Instead, we implemented a **Heuristic Parser** using the PyMuPDF library.

This engine analyzes the raw font metadata stream of the PDF. It calculates the median font size of the document and statistically identifies ”outliers”—text spans that are significantly larger or flagged as bold—as potential section headers.

2.2 Phase 2: Semantic Relevance (NLP Engine)

Objective: Rank the extracted sections based on user intent.

Once headers are identified, we use Natural Language Processing (NLP) to score them. We utilized the `sentence-transformers` library with the `intfloat/e5-small-v2` model. This model converts both the user’s ”Job-to-be-Done” description and the document headers into high-dimensional vector embeddings.

We then calculate the **Cosine Similarity** between the query vector and the header vectors. This allows the system to understand that a query for ”Sustainability” matches a section titled ”Green Energy Initiatives,” even if the exact words differ.

2.3 Phase 3: Output Generation

Objective: Deliver standardized, machine-readable results.

The final phase involves compiling the ranked data into a structured JSON schema. The system filters the top K results (e.g., top 5) and generates a file containing metadata (timestamps, persona details) and the refined text content of the relevant sections.

3 Source Code Description

The codebase is organized into a modular `src/` directory.

3.1 Core Modules

- **`src/parser.py`**: Contains the `PDFParser` class. Uses heuristic logic (font size $>$ median * 1.15 OR bold flag) to extract structural candidates.
- **`src/ranking.py`**: Contains the `RankingEngine` class. Loads the Transformer model and computes cosine similarity scores.
- **`src/output.py`**: Contains the `OutputGenerator` class. Formats the final JSON structure as per the SRS.
- **`src/utils.py`**: Contains text cleaning functions (removing binary garbage, whitespace normalization).

3.2 Execution Scripts

- **`src/main.py`**: CLI entry point using `argparse`.
- **`src/interactive_runner.py`**: Interactive menu entry point reading from `config.json`.

4 Testing Strategy & Validation Approach

To ensure correctness, robustness, and maintainability, we designed a structured testing methodology consisting of Unit Tests (module-level correctness) and Integration Tests (end-to-end behaviour). Instead of validating only output values, the tests verify the underlying assumptions behind the heuristics and semantic ranking process.

4.1 Unit Tests

4.1.1 1. Utility Functions

Goal: Validate that the low-level text-processing logic is reliable.

These tests ensure that:

- Whitespace, noise, and malformed spans are cleaned consistently.
- Title-case and uppercase detection logic behaves predictably.
- Font-weight based bold detection works even with unusual PDF metadata.

This ensures that all downstream modules receive clean, normalized text.

4.1.2 2. Parser Heuristic Tests

Goal: Verify that the structural parser correctly interprets PDF font metadata.

The tests check:

- Headers are detected when font size exceeds the document median.
- Bold text is classified correctly as a heading candidate.
- Wide text spans (large horizontal coverage) are treated as potential section titles.
- Normal body-text spans are correctly rejected.

These tests validate the core assumption of Phase 1: *PDF section titles are statistically distinct from body text*.

4.1.3 3. Semantic Ranking Tests

Goal: Confirm that similarity scoring produces the correct ranking order.

Instead of loading the heavy transformer model, mocks are used to simulate embedding behavior.

The tests validate that:

- Higher similarity values appear earlier in the result list.
- Empty candidate lists return empty outputs.
- The ranking pipeline behaves deterministically when given controlled embeddings.

This ensures that the core semantic logic (cosine similarity-based ranking) functions independently of the model weight files.

4.2 Integration Tests

4.2.1 4. End-to-End Document Processing

Goal: Validate real-world system behaviour across the entire pipeline.

The integration test loads an actual sample PDF and verifies:

1. The parser extracts at least one meaningful heading candidate.
2. The ranking engine returns top-K results that follow descending similarity.
3. The output generator correctly compiles structural and semantic metadata into JSON.
4. A final JSON file is successfully written to disk.

This test simulates a real user workflow and ensures no module-level regressions break pipeline continuity.

4.3 Summary of Test Coverage

Module	Test Type	Purpose
Utilities	Unit	Ensure text cleanup and font-flag logic are correct.
Parser	Unit	Validate heuristic heading identification.
Ranking Engine	Unit	Check correct ordering and empty-input handling.
Full Pipeline	Integration	Ensure end-to-end workflow from PDF to JSON.

This layered testing strategy ensures that each functional block is validated independently as well as collectively, leading to a reliable and maintainable offline document intelligence system.

5 User Manual

5.1 Installation

1. **Unzip/Clone:** Extract the project folder.
2. **Dependencies:** Install required libraries:

```
    pip install -r requirements.txt

```

3. **Data Verification:** Ensure PDF collections are in `data/`.

5.2 Running the Application

5.2.1 Option A: Interactive Mode (Preferred)

Reads from `config.json` and presents a menu.

```
python -m src.interactive_runner
```

5.2.2 Option B: CLI Mode (Manual)

Specify custom paths manually.

```
python -m src.main -i "./data/Collection 1/PDFs" \
                   -p "Travel Planner" \
                   -j "Find travel destinations" \
                   -o "./output/results.json"
```

5.3 Running Tests

The project includes a complete automated test suite. All tests can be executed at once using:

```
python -m unittest discover tests
```

Optionally, an individual test module may be executed for debugging:

```
python -m unittest tests.test_parser
```