Local OCR Search Engine: Technical Report

Viswaz Gummadi

December 26, 2025

1 Introduction

This report documents the development of a privacy-focused, local-first OCR Search Engine. The goal was to build a system capable of extracting text from local images (invoices, screenshot notes, posters) and enabling semantic search without relying on cloud APIs.

2 Phase 1: Initial Architecture

The initial prototype consisted of a simple three-stage pipeline:

1. **OCR**: Using `pytesseract` (Tesseract OCR wrapper) to extract raw text.
2. **Embedding**: Using `sentence-transformers` (*all-MiniLM-L6-v2*) to convert full text blocks into vectors.
3. **Vector Store**: Using `faiss-cpu` with an L2 (Euclidean) distance metric for similarity search.

3 Phase 2: Challenges Experienced

During initial testing, several critical issues were identified:

- **Poor OCR on Stylized Text**: Tesseract failed to read text on noisy backgrounds or stylized fonts (e.g., "NO PAIN NO GAIN" posters).
- **Semantic Dilution**: Embedding the *entire* document text into a single vector caused specific details to get lost ("lost in the middle" phenomenon).
- **Counter-Intuitive Scoring**: The L2 metric meant that a *lower* score was better, which was confusing for ranking logic.

4 Phase 3: Optimization Refinement

To address these challenges, we implemented a series of major architectural upgrades.

4.1 3.1 Advanced Preprocessing (OpenCV)

We introduced an OpenCV-based preprocessing pipeline in `src/core/ocr.py`:

- **Upscaling**: Resizing images by 3x (Cubic Interpolation) to improve character definition.
- **Binary Inverse Thresholding**: Converting images to high-contrast Black & White, specifically targeting posters where text is often lighter than the background.
- **Dilation**: "Thickening" thin fonts so Tesseract can recognize them more easily.

4.2 3.2 Text Chunking

We moved from whole-document embedding to a **Chunking Strategy**:

- **Size**: 500 characters per chunk.
- **Overlap**: 100 characters (to preserve context across boundaries).

This allows the vector database to find specific "needles" in the "haystack" of a document.

4.3 3.3 Metric Switch: Cosine Similarity

We switched the FAISS index from `IndexFlatL2` (Euclidean) to `IndexFlatIP` (Inner Product). By normalizing the embeddings before insertion, Inner Product becomes mathematically equivalent to **Cosine Similarity**.

- **Benefit**: Higher scores now equal better matches (Score 1.0 = Exact Match).
- **Bug Fix**: We corrected the sorting logic in `main.py` to sort results in **Descending** order.

5 Phase 4: Verification & Stress Testing

To ensure robustness, we created a comprehensive stress test suite (`tests/stress_test.py`) that verifies:

1. **Chunking Logic**: Ensuring text splits happen at correct boundaries.
2. **Sorting Integrity**: Confirming that a closer vector match yields a technically higher score.
3. **Mass Indexing**: Simulating the indexing of 1000 documents to verify database persistence and performance.

6 Conclusion

The final system is a robust, highly accurate local search engine. It successfully handles noisy images via Computer Vision, retrieves granular details via Chunking, and ranks results intuitively using Cosine Similarity.