Plagiarism Detection

Dataset Splitting:

Out of the given samples, 80% of the resumes are fed to the Vector DB and 20% are stored for testing.

Text Extraction:

Approach 1:

Unstructured.io is employed to convert resumes into Markdown format. Markdown is chosen for its semantic structure, which organizes documents into clear sections and headers — beneficial for comparing resumes at a section level.
Chunking is handled via LangChain's MarkdownTextSplitter, and the resulting segments are passed to the Sentence Transformer model intfloat/multilingual-e5-large-instruct (running locally) to generate embeddings. These vectors are then stored in **QdrantDB**, selected for its open-source nature, Approximate Nearest Neighbor (ANN) indexing, and in-built vector normalization (optimizing cosine similarity computations).

For similarity checks, test resumes are also parsed using Unstructured.io and chunked accordingly to ensure consistent comparison against the indexed database.

Approach 2:

An alternative pipeline utilizes **LibreOffice** (installed locally) to convert .doc and .docx files into PDFs, which are then processed with **PyMuPDF4LLM** to extract structured Markdown. This output is chunked using the same LangChain utility, and embeddings are generated and stored as in Approach 1. Here, Unstructured is only used as a fallback option.

While both approaches are functionally similar in architecture, Approach 2 is deemed more performant and scalable for production use due to the following factors:

1. **Speed**:
    - PyMuPDF4LLM offers fast text extraction (~0.1–0.3s/page), significantly outperforming Unstructured.io (0.5–2.5s/page).

  - LibreOffice reuses processes, enabling rapid document conversion (<1s/file), whereas Unstructured.io incurs consistent overheads from ML/OCR pipelines.

2. **Output Optimization**:
    - PyMuPDF4LLM's Markdown output is streamlined for LLM/RAG workflows, reducing the need for downstream processing.
    - Unstructured.io offers robust layout and metadata extraction, but these are less critical for standardized resume formats
3. Feasibility:
    - Approach 2 is approximately **2.5x faster** on average, making it more suitable for high-throughput environments.

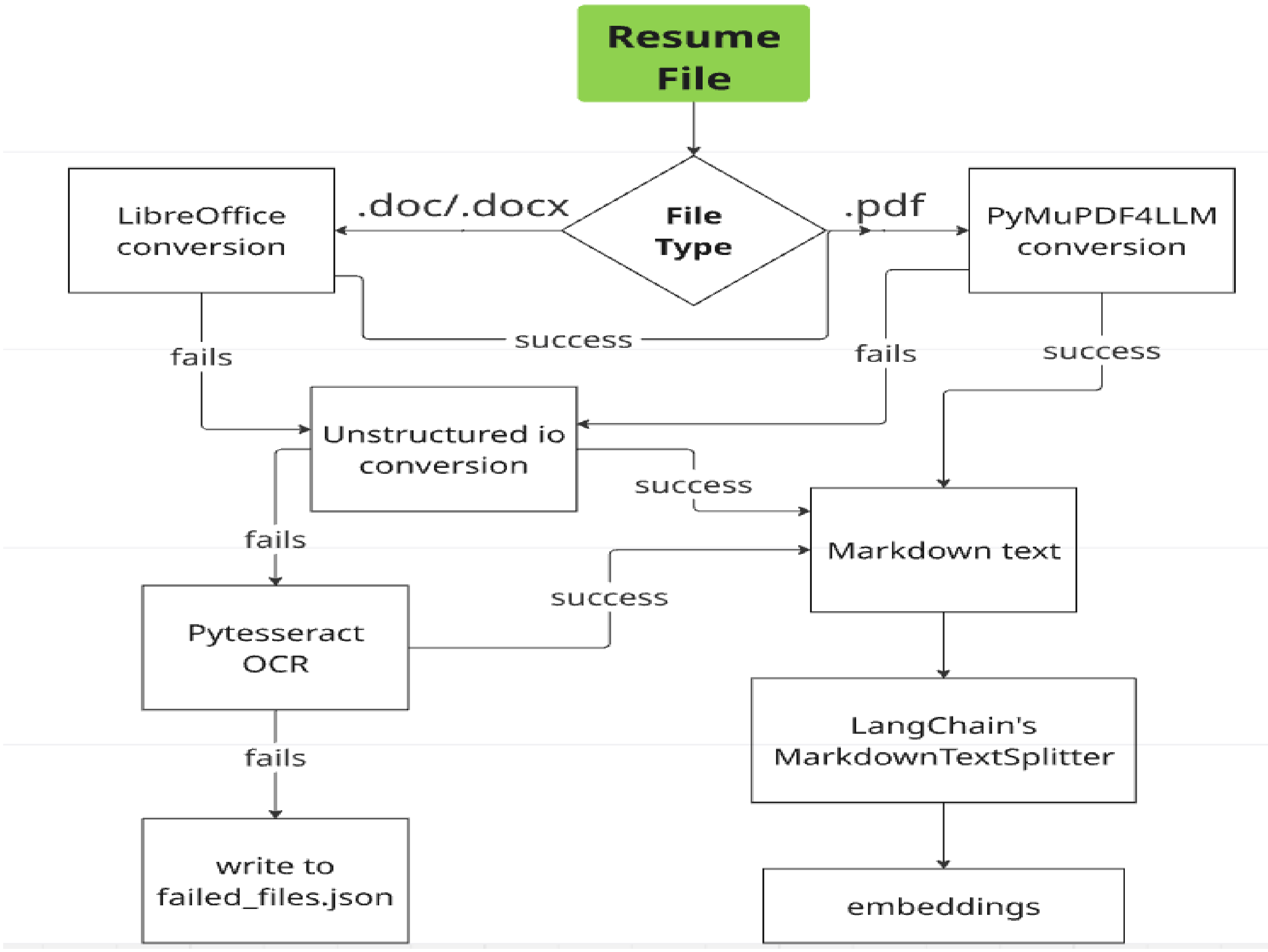
Resume Loader Module

The Resume Loader Module (pipeline.py) is responsible for uploading reference resumes to a Qdrant vector database for use in plagiarism checks. It processes .docx and .pdf files, extracts text, generates embeddings, and stores them with metadata.

Approach:

- **Text Extraction**: Convert .docx files to PDF using LibreOffice, then extract text as markdown using libraries like pymupdf4llm. For PDFs, attempt text extraction first, falling back to OCR (tesseract) for image-based files.
- **Embedding Generation**: Use a SentenceTransformer model (intfloat/multilingual-e5-large-instruct) to create vector embeddings of resume chunks, which are stored in Qdrant with source file metadata.
- **Error Handling**: Improved logging to track failed uploads and added retry logic for deleting temporary PDFs to prevent file accumulation.
- **Optimization**: Focused on processing only reference files, skipping redundant operations to reduce overhead.

Fallback Methods used:



Plagiarism Detection Module

The Plagiarism Detection Module (plagiarism_detector.py) analyzes input resumes for similarities against reference resumes in Qdrant, extracting contact information and computing similarity scores.

Approach:

- **Text Processing**: Extract text using the same utilities as the loader module, ensuring consistency. Split text into chunks for analysis, filtering out non-critical sections like hobbies.
- **Contact Extraction**: Identify emails and phone numbers using regex, with OCR fallback for scanned documents. Enhanced validation to prevent errors when file paths are invalid.

- **Similarity Scoring**: Generate embeddings for input resume chunks and query Qdrant for matches. Compute composite scores based on cosine similarity, TF-IDF, fuzzy matching, and n-grams to produce an overall_score (average chunk similarity) and flagged_score (plagiarized chunks).
- **Filtering Refinement**: Adjusted chunk filtering to allow technical terms (e.g., "programming languages") while excluding irrelevant content (e.g., "hobbies:"), preventing all chunks from being discarded.
- **Error Mitigation**: Added robust input validation and error handling to manage empty text or invalid paths, ensuring graceful failure.

Shared Utilities

Both modules rely on common_utils.py for shared functions like extract_text_from_file, convert_doc_to_pdf, and Qdrant operations. We enhanced these utilities with better logging, type checking, and retry mechanisms to ensure reliability across diverse file formats.

Metrics Used:

- **Cosine Similarity**

  Highly effective for detecting semantic similarity, especially when using modern language models (like Sentence Transformers), as it captures both word usage and context.

- **TF-IDF Similarity**

  Good for comparing documents based on word frequency and uniqueness, but less sensitive to semantic nuance compared to modern embeddings.

- **Fuzzy Similarity (Levenshtein Distance)**

  Measures how many single-character edits. Useful for detecting verbatim or near-verbatim copying, especially when texts are only slightly altered.

- **N-gram Similarity**

  Effective for detecting copied or paraphrased text at the phrase level, as it is robust to minor changes in word order or wording.

Reasoning for the Weights:

- **Cosine Similarity (Weight: 0.4)**

  Cosine similarity from embeddings is weighted the highly as this ensures that semantic similarity is prioritized, which is crucial for detecting both direct and paraphrased plagiarism. It is observed that too much of it's weight increases false positives.

- **N-gram Similarity (Weight: 0.4)**

  Good at catching paraphrasing and word overlap between grams. 3-grams have been given the highest priority as it is observed in research that this yields best outputs.

Ex> "I love eating ice cream" is split for 3-grams as:

1. **I love eating**
2. **love eating ice**
3. **eating ice cream**

- **TF-IDF Similarity (Weight: 0.15)**

  TF-IDF is less sensitive to context and nuance compared to embeddings, but it is still useful for comparing documents based on word usage. Its lower weight reflects its secondary role in the composite score.

- **Fuzzy Similarity (Weight: 0.05)**

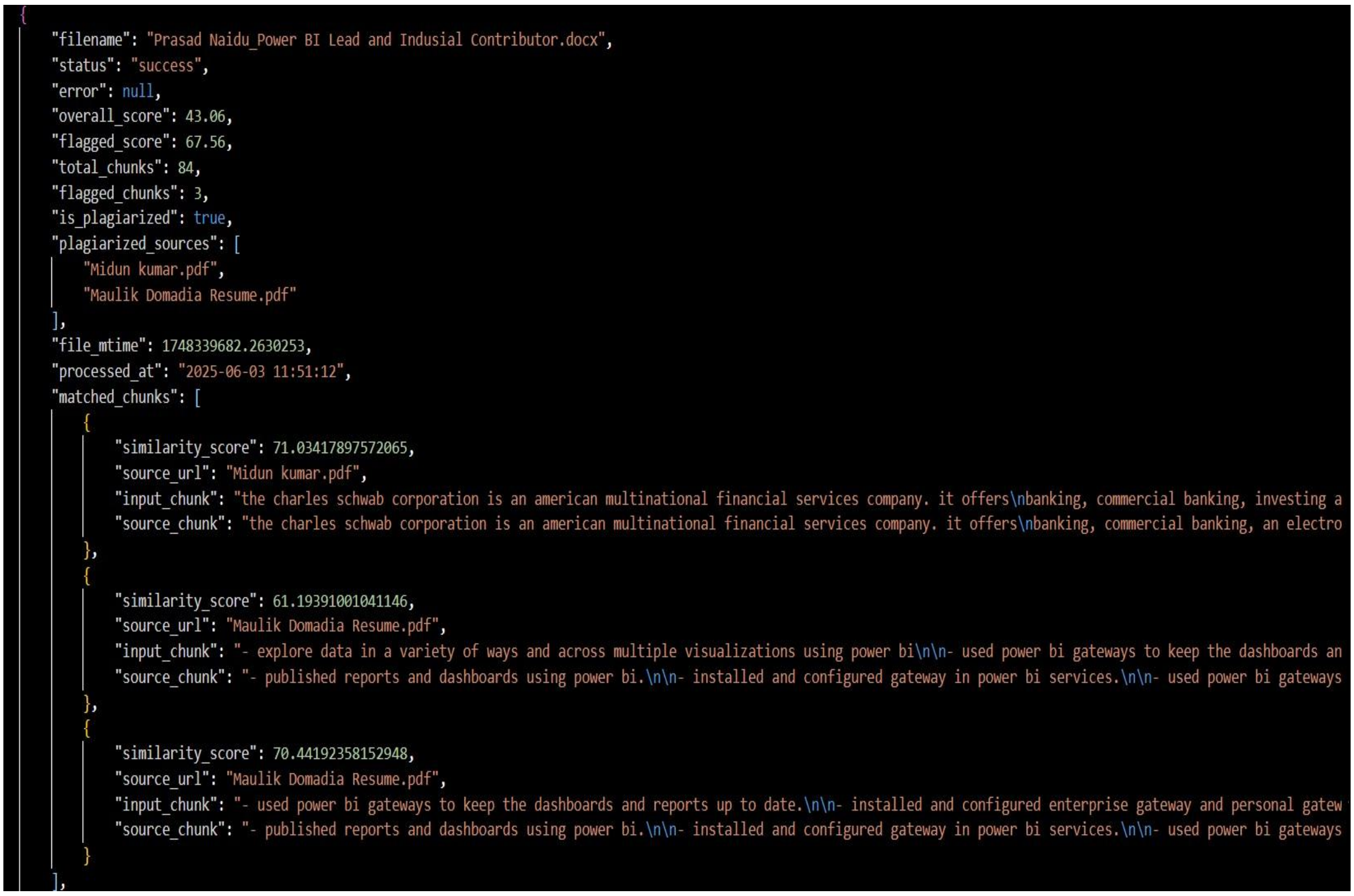
  Levenshtein (fuzzy) similarity is mainly useful for detecting verbatim or near-verbatim copying. Since most plagiarism involves some degree of rewording, this measure is less important and thus given the lowest weight.

Post-Detection Actions

- If a resume is flagged for plagiarism:
  - **Phone numbers** are extracted using regex and the phonenumbers Python library. o **Email addresses** are extracted using regular expressions.
  - Both are stored in a structured JSON format for further action or auditing.

Optimization & File Handling

- A cache of already processed files is maintained in processedfiles.json.
- Modification timestamps (mtime) are used to skip redundant processing, significantly reducing execution time for repeated runs.



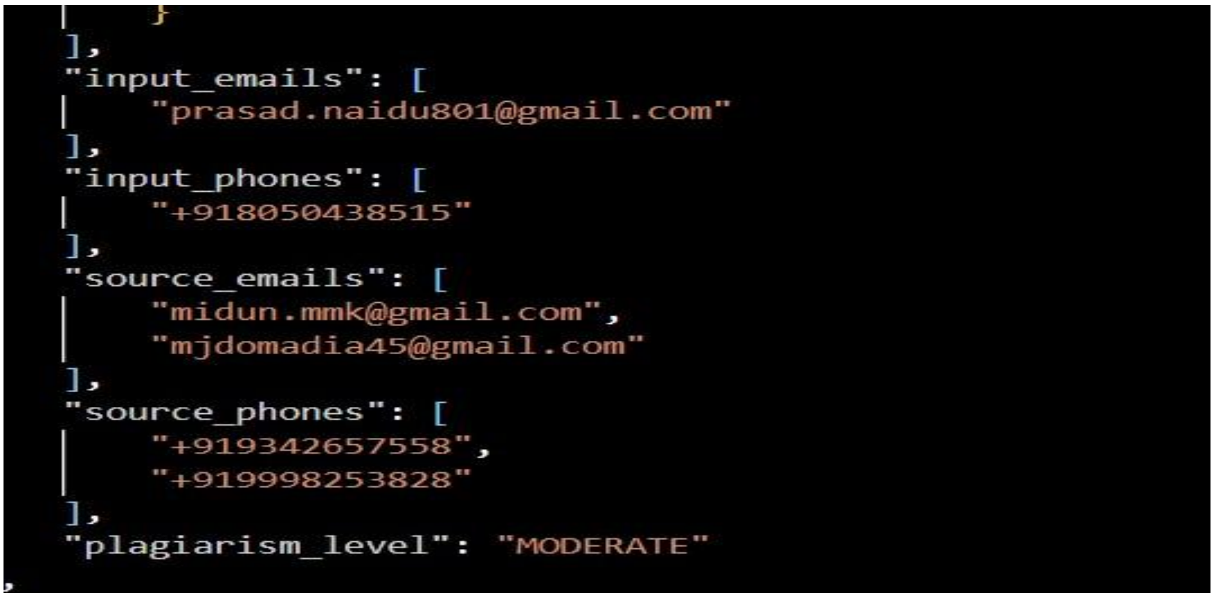
```
{
    "filename": "Prasad Naidu_Power BI Lead and Indusial Contributor.docx",
    "status": "success",
    "error": null,
    "overall_score": 43.06,
    "flagged_score": 67.56,
    "total_chunks": 84,
    "flagged_chunks": 3,
    "is_plagiarized": true,
    "plagiarized_sources": [
        "Midun kumar.pdf",
        "Maulik Domadia Resume.pdf"
    ],
    "file_mtime": 1748339682.2630253,
    "processed_at": "2025-06-03 11:51:12",
    "matched_chunks": [
        {
            "similarity_score": 71.03417897572065,
            "source_url": "Midun kumar.pdf",
            "input_chunk": "the charles schwab corporation is an american multinational financial services company. it offers\nbanking, commercial banking, investing a
            "source_chunk": "the charles schwab corporation is an american multinational financial services company. it offers\nbanking, commercial banking, an electro
        },
        {
            "similarity_score": 61.19391001041146,
            "source_url": "Maulik Domadia Resume.pdf",
            "input_chunk": "- explore data in a variety of ways and across multiple visualizations using power bi\n\n- used power bi gateways to keep the dashboards an
            "source_chunk": "- published reports and dashboards using power bi.\n\n- installed and configured gateway in power bi services.\n\n- used power bi gateways
        },
        {
            "similarity_score": 70.44192358152948,
            "source_url": "Maulik Domadia Resume.pdf",
            "input_chunk": "- used power bi gateways to keep the dashboards and reports up to date.\n\n- installed and configured enterprise gateway and personal gatew
            "source_chunk": "- published reports and dashboards using power bi.\n\n- installed and configured gateway in power bi services.\n\n- used power bi gateways
        }
    ],
```

Querying the Vector DB

- Qdrant's query_points API is used to fetch the **top 5 nearest embeddings** per resume.
- These are used to evaluate similarity and confirm potential plagiarism cases with high confidence.

<u>Output:</u>

The output/ results are stored in results folder, inside **"plagiarism_results.json"** file. It contains all the metadata along with it's **overall_score** that shows the similarity the given resume holds againstthe existing resume database. It also has **flagged_score** that shows the percentage of the chunks that are actually plagiarized out of the similar chunks. Detailed chunk similarity comparisons are also stored in this json file. The plagiarism level is attached in the file. For extra metadata, the input resume candidate’s phone number and email along with the similar source file ‘s candidates emails and phone numbers are append to the plagiarized files result.



```
        }
    ],
    "input_emails": [
        "prasad.naidu801@gmail.com"
    ],
    "input_phones": [
        "+918050438515"
    ],
    "source_emails": [
        "midun.mmk@gmail.com",
        "mjdomadia45@gmail.com"
    ],
    "source_phones": [
        "+919342657558",
        "+919998253828"
    ],
    "plagiarism_level": "MODERATE"
,
```

- Addressed the issue where, a candidate’s file is labelled as “plagiarized” against their own resume with different filename by checking for same email and phone number match.

- Also addressed the issue where, any two candidate’s files were categorized as plagiarized because of matching hobbies/interests/languages by ignoring these keywords while checking for chunk similarity

- Used **RotatingFileHandler** to manage log files efficiently by limiting each log file’s size and maintaining a fixed number of backup files. In this project, it ensures logs are rotated after reaching 5 MB, retaining the three most recent backups to prevent disk bloat while preserving recent activity.

- **PIL (Python Imaging Library)** modules like Image, ImageEnhance, and ImageOps were used to preprocess scanned resume images—enhancing contrast and readability—to improve OCR accuracy for extracting contact information such as phone numbers and email addresses

1. Contextual Word Embedding Similarity (0.22)
Captures deep semantic relationships by comparing contextualized embeddings generated from transformer models. This component excels at identifying paraphrased or reworded passages that preserve meaning despite lexical changes[1].
2. BLEURT Score (0.18)
Measures fine-grained semantic similarity using a BERT-based metric trained on human judgments. BLEURT effectively correlates with human assessments of paraphrase quality, boosting detection of nuanced text modifications[2].
3. Lexical Fingerprint (0.16)
Implements the winnowing algorithm to generate robust k-gram hashes for exact and near-exact match detection. Lexical fingerprinting excels at flagging verbatim reuse and template copying common in resume plagiarism[3].
4. Semantic Role Similarity (0.14)
Compares semantic role label structures (e.g., agent, action, object) between sentences, enabling detection of meaning preservation despite word substitutions. This approach uncovers idea-level plagiarism by focusing on argument patterns rather than surface text[4].
5. Cosine Similarity (0.10)
Computes vector similarity on TF-IDF representations to detect high lexical overlap. Cosine similarity serves as a reliable baseline for statistical term matching, quickly identifying bulk text reuse[5].
6. Structural Similarity (0.08)
Analyzes stopword n-gram or syntactic patterns to capture organizational likeness. Structural similarity flags resumes with identical section layouts or repeated formatting choices, even when content words differ[6].
7. TF-IDF Similarity (0.06)
Assesses term importance across documents by weighting rare but significant resume terms more heavily. TF-IDF similarity aids in highlighting reuse of industry-specific skills or project descriptions that appear infrequently in general corpora[5].
8. N-gram Similarity (0.03)
Measures overlap of contiguous word sequences (e.g., bi-grams, tri-grams). N-gram similarity effectively catches localized copying of short phrases and common resume bullet points[6].
9. Stylometric Similarity (0.02)
Evaluates writing style features such as average sentence length, punctuation patterns, and function-word frequencies. Stylometric similarity helps detect shifts in authorial style indicative of pasted sections from other sources[7].
10. Word Order Similarity (0.01)

Quantifies alignment of word sequences to identify passages with preserved syntax. Though low-weighted, this component contributes to detecting rearranged text segments where meaning order remains intact[6].