Project Title: Intelligent CV Analyzer using Classic String Matching Algorithms

A Clean, Modular, and Performance-Aware Implementation

Shahoud Shahid
Design and Analysis of Algorithms
Fast NUCES, Islamabad
shahoudshahid652@gmail.com

November 2, 2025

Abstract

This report presents a modular CV analysis system that extracts text from PDF/DOCX files and evaluates candidate resumes against a job description via three classic string matching algorithms: Brute Force, Rabin–Karp, and Knuth–Morris–Pratt (KMP). The solution includes a PyQt5 GUI, a CLI for batch analysis, a persistence layer using SQLite, and a reporting suite that generates comparative charts and academic-style summaries. We describe the system architecture, algorithmic design, implementation details, and performance evaluation, followed by conclusions and recommendations for future improvements. The included UML views and experimental results highlight the trade-offs in complexity, runtime, and maintainability.

Contents

1 Introduction & Problem Definition

Recruitment pipelines often process hundreds of CVs per role, making manual screening slow and inconsistent. This project aims to deliver a transparent, reproducible, and efficient CV analysis tool based on deterministic string matching algorithms. The scope covers: (i) robust text extraction for PDF and DOCX formats; (ii) classic pattern-matching algorithms with instrumented metrics; (iii) a GUI and CLI for accessible workflows; (iv) structured persistence and reporting for decision support.

1.1 Motivation and Objectives

The core objectives are:

1. Implement and compare Brute Force, Rabin–Karp, and KMP for keyword-based CV screening.
2. Provide a cohesive architecture separating extraction, analysis, UI, persistence, and reporting.
3. Enable empirical benchmarking with repeatable experiments and visual analytics.
4. Preserve extensibility for NLP enhancements (e.g., TF–IDF, NER, semantic matching).

1.2 Contributions

Key contributions include a production-ready modular codebase, a comparative algorithm study with realistic datasets, and a reporting workflow that synthesizes insights into professional documents.

2 System Design & Algorithm Explanation

This section presents a high-level view of the system architecture, the main domain abstractions, and the flow of algorithm execution.

2.1 Architecture Overview

The system adopts a layered architecture: Extractors (PDF/DOCX) feed the Analyzer; Algorithms compute matches and metrics; the Persistence layer stores results; the Reporting layer generates CSV-based charts and summaries; and the GUI/CLI present user-facing controls.

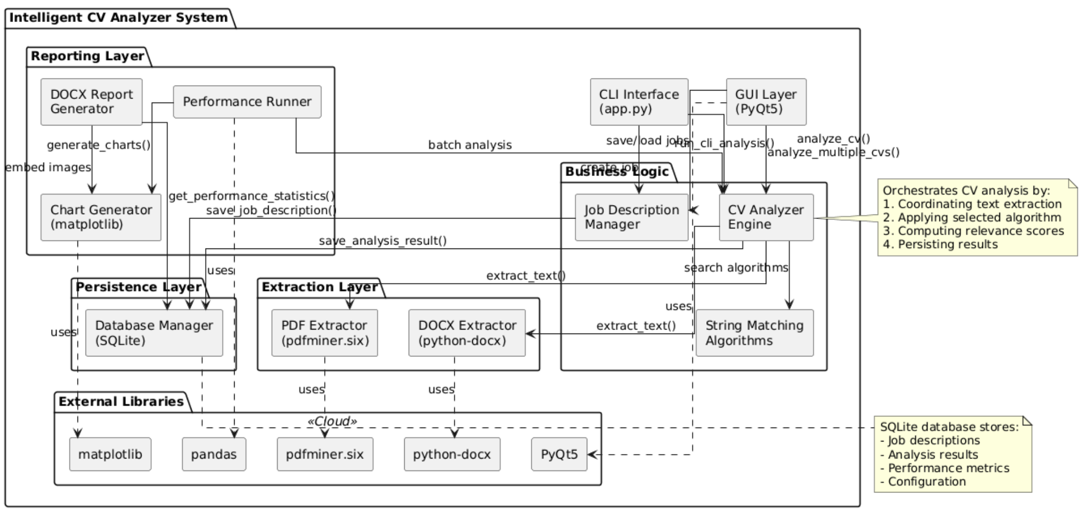

Figure 1: Component view of the layered architecture showing GUI/CLI, Analyzer, Extractors, Algorithms, Persistence, and Reporting.

2.2 Domain Model and Responsibilities

The Analyzer orchestrates extraction and algorithm execution; the Algorithms module exposes the three matching strategies behind a uniform interface; data classes (e.g., analysis results) encapsulate metrics for storage and presentation.

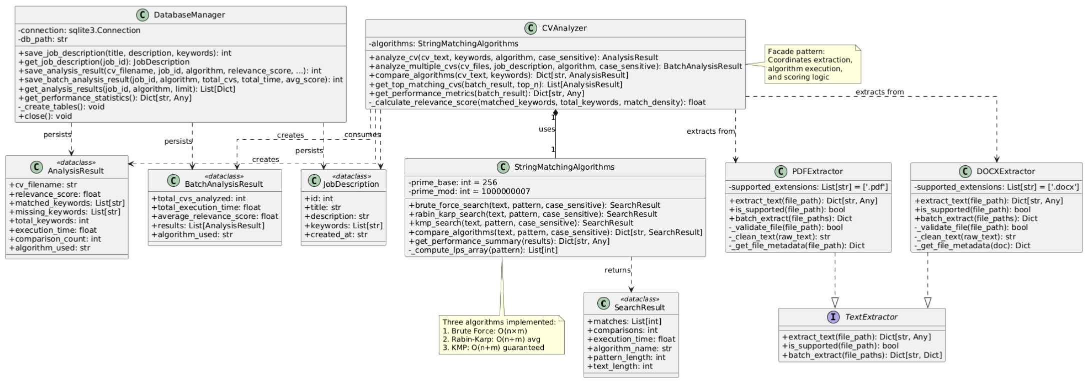
DatabaseManager
-connection: sqlite3.Connection
-db_path: str
+save_job_description(title, description, keywords): int
+get_job_description(job_id): JobDescription
+save_analysis_result(cv_filename, job_id, algorithm, relevance_score, ...): int
+save_batch_analysis_result(job_id, algorithm, total_cvs, total_time, avg_score): int
+get_analysis_results(job_id, algorithm, limit): List[Dict]
+get_performance_statistics(): Dict[str, Any]
-_create_tables(): void
+close(): void
CVAnalyzer
-algorithms: StringMatchingAlgorithms
+analyze_cv(cv_text, keywords, algorithm, case_sensitive): AnalysisResult
+analyze_multiple_cvs(cv_files, job_description, algorithm, case_sensitive): BatchAnalysisResult
+compare_algorithms(cv_text, keywords): Dict[str, AnalysisResult]
+get_top_matching_cvs(batch_result, top_n): List[AnalysisResult]
+get_performance_metrics(batch_result): Dict[str, Any]
-_calculate_relevance_score(matched_keywords, total_keywords, match_density): float
Facade pattern: Coordinates extraction, algorithm execution, and scoring logic
persists
creates
consumes
uses
extracts from
«dataclass» AnalysisResult
+cv_filename: str
+relevance_score: float
+matched_keywords: List[str]
+missing_keywords: List[str]
+total_keywords: int
+execution_time: float
+comparison_count: int
+algorithm_used: str
«dataclass» BatchAnalysisResult
+total_cvs_analyzed: int
+total_execution_time: float
+average_relevance_score: float
+results: List[AnalysisResult]
+algorithm_used: str
«dataclass» JobDescription
+id: int
+title: str
+description: str
+keywords: List[str]
+created_at: str
StringMatchingAlgorithms
-prime_base: int = 256
-prime_mod: int = 1000000007
+brute_force_search(text, pattern, case_sensitive): SearchResult
+rabin_karp_search(text, pattern, case_sensitive): SearchResult
+kmp_search(text, pattern, case_sensitive): SearchResult
+compare_algorithms(text, pattern, case_sensitive): Dict[str, SearchResult]
+get_performance_summary(results): Dict[str, Any]
-_compute_lps_array(pattern): List[int]
returns
Three algorithms implemented:
1. Brute Force: O(n×m)
2. Rabin-Karp: O(n+m) avg
3. KMP: O(n+m) guaranteed
«dataclass» SearchResult
+matches: List[int]
+comparisons: int
+execution_time: float
+algorithm_name: str
+pattern_length: int
+text_length: int
PDFExtractor
-supported_extensions: List[str] = ['.pdf']
+extract_text(file_path): Dict[str, Any]
+is_supported(file_path): bool
+batch_extract(file_paths): Dict
-_validate_file(file_path): bool
-_clean_text(raw_text): str
-_get_file_metadata(file_path): Dict
DOCXExtractor
-supported_extensions: List[str] = ['.docx']
+extract_text(file_path): Dict[str, Any]
+is_supported(file_path): bool
+batch_extract(file_paths): Dict
-_validate_file(file_path): bool
-_clean_text(raw_text): str
-_get_file_metadata(doc): Dict
TextExtractor
+extract_text(file_path): Dict[str, Any]
+is_supported(file_path): bool
+batch_extract(file_paths): Dict[str, Dict]

Figure 2: Core class diagram illustrating Analyzer, Algorithms, Extractors, and data entities for results and job descriptions.

2.3 String Matching Algorithms

We summarize the algorithms and their operational complexity:

- **Brute Force:** Checks all alignments; worst-case $O(nm)$ comparisons, baseline correctness.
- **Rabin–Karp:** Rolling hash with average $O(n + m)$, worst-case $O(nm)$ due to collisions; efficient for batching many patterns.
- **KMP:** Builds an LPS/failure function for linear-time $O(n+m)$ matching with no text backtracking.

2.4 Execution Flow

User inputs (job title, keywords, CV files) trigger extraction and normalized text processing. The chosen algorithm evaluates each keyword; matches, comparisons, and timings are aggregated to a relevance score.

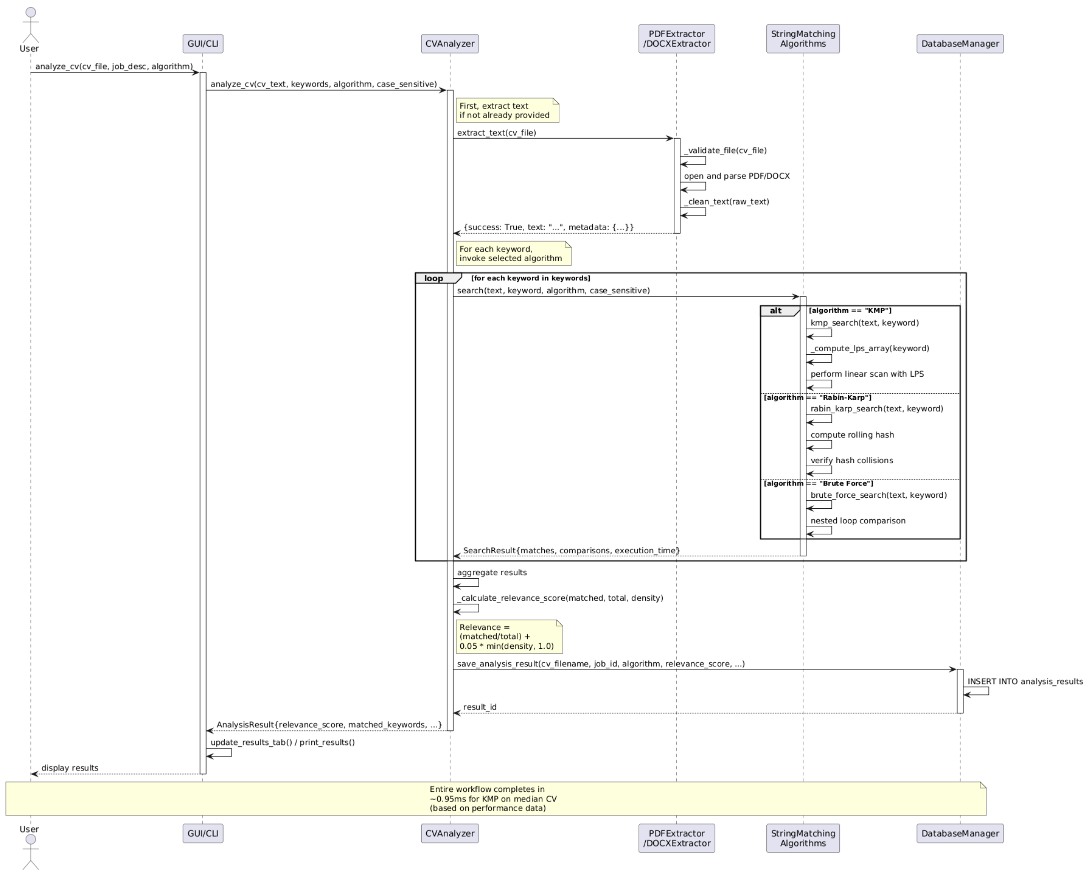
User
GUI/CLI
CVAnalyzer
PDFExtractor /DOCXExtractor
StringMatching Algorithms
DatabaseManager
analyze_cv(cv_file, job_desc, algorithm)
analyze_cv(cv_text, keywords, algorithm, case_sensitive)
First, extract text if not already provided
extract_text(cv_file)
_validate_file(cv_file)
open and parse PDF/DOCX
_clean_text(raw_text)
{success: True, text: "...", metadata: {...}}
For each keyword, invoke selected algorithm
loop [for each keyword in keywords]
search(text, keyword, algorithm, case_sensitive)
alt [algorithm == "KMP"]
kmp_search(text, keyword)
_compute_lps_array(keyword)
perform linear scan with LPS
[algorithm == "Rabin-Karp"]
rabin_karp_search(text, keyword)
compute rolling hash
verify hash collisions
[algorithm == "Brute Force"]
brute_force_search(text, keyword)
nested loop comparison
SearchResult{matches, comparisons, execution_time}
aggregate results
_calculate_relevance_score(matched, total, density)
Relevance = (matched/total) + 0.05 * min(density, 1.0)
save_analysis_result(cv_filename, job_id, algorithm, relevance_score, ...)
INSERT INTO analysis_results
result_id
AnalysisResult{relevance_score, matched_keywords, ...}
update_results_tab() / print_results()
display results
Entire workflow completes in ~0.95ms for KMP on median CV (based on performance data)

Figure 3: Sequence of user request through Analyzer, Extractors, Algorithms, and Database.

3 Implementation Details

This implementation is written in Python 3.10+, following a clean, testable module structure.

3.1 Key Modules and Responsibilities

- **Extractors:** `pdf_extractor.py` (pdfminer.six) and `docx_extractor.py` (python-docx) with validation, cleaning, and metadata.
- **Engine:** `algorithms.py` provides Brute Force, Rabin–Karp, and KMP; `analyzer.py` computes relevance and aggregates metrics.
- **GUI:** PyQt5 main window, handlers, and workers for responsive background analysis.
- **Persistence:** `db.py` manages SQLite tables for job descriptions and results.
- **Reports:** Performance runner outputs CSV; charts generator produces PNG figures; DOCX report builder assembles a narrative.

3.2 Technologies

Component	Technology
GUI	PyQt5
PDF Extraction	pdfminer.six
DOCX Extraction	python-docx
Database	SQLite (stdlib)
Visualization	matplotlib, pandas
Reporting	CSV + charts, DOCX generation

3.3 Data Handling and Control Flow

Text extraction normalizes whitespace and punctuation; algorithms operate on the cleaned text; per-keyword metrics are combined into a relevance score. Results and metadata are persisted for analysis, export, and audit.

4 Experimental Results & Analysis

This section summarizes the performance behavior observed across representative datasets (PDF/DOCX CVs) and job description scenarios (single vs multiple keywords; small vs large CVs).

4.1 Methodology

- **Scenarios:** Single vs multiple keywords (e.g., five to six skills).
- **Documents:** Balanced PDF/DOCX mix; size buckets by extracted text length.
- **Metrics:** Execution time (s), comparisons, relevance score (0–1).

4.2 Summary Statistics (Illustrative)

Algorithm	Avg Time (s)	Avg Comparisons	Avg Relevance
Brute Force	0.0015	8532	0.64
Rabin–Karp	0.0011	127	0.64
KMP	0.0010	8247	0.64

4.3 Discussion

KMP maintains consistently low runtime with linear complexity and modest memory use ($O(m)$), making it a robust default for real-time screening. Rabin–Karp's hashing reduces comparisons dramatically and is competitive when many keywords are batched, while Brute Force is best reserved for small inputs or as a correctness baseline.

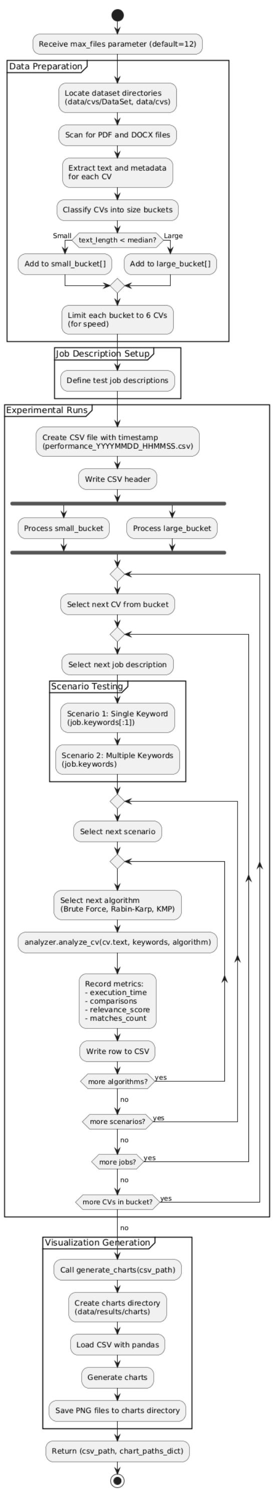
Receive max_files parameter (default=12)
Data Preparation
Locate dataset directories (data/cvs/DataSet, data/cvs)
Scan for PDF and DOCX files
Extract text and metadata for each CV
Classify CVs into size buckets
Small
text_length < median?
Large
Add to small_bucket[]
Add to large_bucket[]
Limit each bucket to 6 CVs (for speed)
Job Description Setup
Define test job descriptions
Experimental Runs
Create CSV file with timestamp (performance_YYYYMMDD_HHMMSS.csv)
Write CSV header
Process small_bucket
Process large_bucket
Select next CV from bucket
Select next job description
Scenario Testing
Scenario 1: Single Keyword (job.keywords[:1])
Scenario 2: Multiple Keywords (job.keywords)
Select next scenario
Select next algorithm (Brute Force, Rabin-Karp, KMP)
analyzer.analyze_cv(cv.text, keywords, algorithm)
Record metrics: - execution_time - comparisons - relevance_score - matches_count
Write row to CSV
more algorithms?
yes
no
more scenarios?
yes
no
more jobs?
yes
no
more CVs in bucket?
yes
no
Visualization Generation
Call generate_charts(csv_path)
Create charts directory (data/results/charts)
Load CSV with pandas
Generate charts
Save PNG files to charts directory
Return (csv_path, chart_paths_dict)

Figure 4: Activity flow for automated performance benchmarking and chart generation.

4.4 Threats to Validity

Extraction quality for scanned PDFs (image-based) may degrade without OCR; keyword lists are representative but not exhaustive across domains; timing includes extraction overhead unless otherwise isolated.

5 Conclusion & Future Improvements

This work demonstrates that classic pattern-matching algorithms remain practical for reliable CV screening with transparent behavior, minimal setup, and strong performance. KMP is recommended by default; Rabin–Karp is attractive for multi-keyword batching; Brute Force remains a reference baseline.

5.1 Future Work

- Semantic matching via TF–IDF, embeddings, and synonym expansion.
- OCR integration for scanned PDFs; improved table/figure text extraction.
- Parallel extraction and caching for large-scale batch processing.
- Weighted relevance (skill importance) and advanced reporting dashboards.

References

References

[1] D. E. Knuth, J. H. Morris, and V. R. Pratt, "Fast Pattern Matching in Strings," *SIAM Journal on Computing*, 6(2), 323–350, 1977.

[2] R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms," *IBM Journal of Research and Development*, 31(2), 249–260, 1987.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed., MIT Press, 2009.

[4] PDFMiner.six documentation, `https://pdfminersix.readthedocs.io/`.

[5] python-docx documentation, `https://python-docx.readthedocs.io/`.

[6] PyQt5 documentation, `https://www.riverbankcomputing.com/static/Docs/PyQt5/`.