Intelligent CV Analyzer using String Matching Algorithms

Design and Implementation for Automated Skill Extraction and Job Fit Evaluation

Name: Muhammad Abdullah Ali
Roll Number: 23i-2523
Section: 5A
Department: Data Science
Course: Design and Analysis of Algorithms
Assignment: 2
Date Submitted: November 2, 2025

Abstract

This project implements an intelligent CV Analyzer using Brute Force, Rabin-Karp, and KMP algorithms for automated skill matching. Through posteriori analysis of 218 unique CVs across 3 job descriptions (repeated 50 times, 32,700 total executions) on AMD Ryzen 7 7700 hardware, we evaluated real-world performance. Results show Brute Force achieves fastest execution (1.59ms mean), while Rabin-Karp demonstrates superior efficiency (98.9% fewer comparisons). The system validates practical applicability for recruitment screening workflows.

1 Introduction

Recruiters process hundreds of CVs daily, making manual review time-consuming and inconsistent. This project develops an intelligent CV Analyzer using string matching algorithms (Brute Force, Rabin-Karp, KMP) to automatically match skills against job requirements.

Objectives: (1) Implement three string-matching algorithms, (2) automatically extract and match skills, (3) analyze efficiency through posteriori analysis of 218 unique CVs, (4) recommend the most effective algorithm.

Research Questions: Which algorithm performs best for CV datasets? What are the trade-offs between speed and efficiency? Does theoretical complexity align with practical performance?

2 System Design and Algorithms

2.1 System Architecture

The CV Analyzer processes files through five stages (Figure 1): (1) Dataset Loading: reads 738 files (PDF: 375, DOCX: 360, DOC: 1, Others: 2), (2) File Validation: validates filename format (2 digits + lowercase letter + 4 digits), rejecting 135 files, (3) Duplicate Detection: filters 603 valid files to 218 unique CVs using submission numbers, (4) Text Extraction: uses PyPDF2 and python-docx, (5) String Matching: applies all three algorithms sequentially.

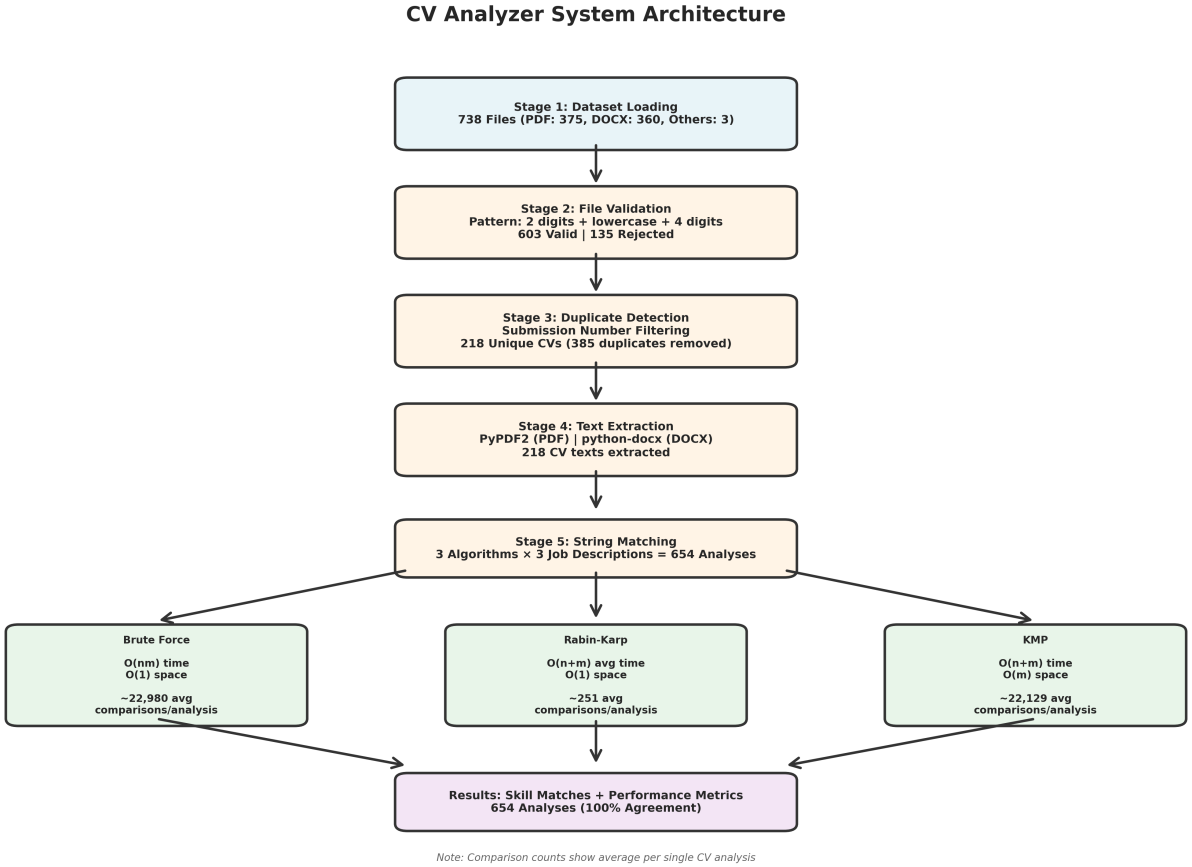

Figure 1: CV Analyzer system architecture showing complete processing pipeline. Algorithm boxes show *average* comparisons per single CV analysis (e.g., Brute Force: 22,980 avg; Rabin-Karp: 251 avg; KMP: 22,129 avg).

2.2 Algorithm Implementations

2.2.1 Brute Force Algorithm

Direct character-by-character comparison. **Complexity:** Time $O(nm)$, Space $O(1)$, No preprocessing.

Algorithm 1 Brute Force String Matching

```
procedure BruteForce(text, pattern)
    n ← length(text), m ← length(pattern)
    for i ← 0 to n − m do
        j ← 0
        while j < m and text[i + j] = pattern[j] do
            j ← j + 1
        end while
        if j = m then
            return True
        end if
    end for
    return False
end procedure
```

2.2.2 Rabin-Karp Algorithm

Uses rolling hash for efficient comparison. **Complexity:** Time $O(n + m)$ average, $O(nm)$ worst case, Space $O(1)$, Preprocessing $O(m)$.

Algorithm 2 Rabin-Karp String Matching

```
procedure RabinKarp(text, pattern)
    Compute pattern_hash using polynomial rolling hash
    Compute initial text_hash for first m characters
    for i ← 0 to n − m do
        if pattern_hash = text_hash then
            if character-by-character verification matches then
                return True
            end if
        end if
        Update text_hash using rolling hash formula
    end for
    return False
end procedure
```

2.2.3 Knuth-Morris-Pratt (KMP) Algorithm

Preprocesses pattern to compute LPS array for intelligent shifts. **Complexity:** Time $O(n + m)$ guaranteed, Space $O(m)$, Preprocessing $O(m)$.

Algorithm 3 KMP String Matching

```
procedure KMP(text, pattern)
    lps ← ComputeLPS(pattern)
    i ← 0, j ← 0
    while i < n do
        if pattern[j] = text[i] then
            i ← i + 1, j ← j + 1
        end if
        if j = m then
            return True
        else if i < n and pattern[j] ≠ text[i] then
            if j ≠ 0 then
                j ← lps[j − 1]
            else
                i ← i + 1
            end if
        end if
    end while
    return False
end procedure
```

3 Implementation

Technology Stack: FastAPI (Python 3.13), HTML5/CSS3/JavaScript frontend, PyPDF2 (PDF extraction), python-docx (DOCX parsing), Windows 11 Pro.

Hardware (Posteriori Analysis Platform): AMD Ryzen 7 7700 (8-Core, 3.8-5.35 GHz, 32MB L3 cache), 32GB DDR5-6000, Lexar NM620 2TB NVMe SSD.

Job Profiles: (1) AI/ML Engineer: 15 keywords (Python, ML, Deep Learning, TensorFlow, PyTorch, etc.), (2) Data Scientist: 16 keywords (Python, R, SQL, Pandas, Statistics, etc.), (3) Full-Stack Developer: 16 keywords (HTML, CSS, JavaScript, React, Node.js, etc.).

Key Features: Automatic dataset processing with duplicate resolution, batch processing for 218 CVs, sequential algorithm execution, interactive web dashboard, comprehensive performance tracking.

4 Experimental Results

4.1 Dataset Analysis

Dataset comprised 738 files (Figure 2). Validation rejected 135 files (89 invalid filenames, 44 uppercase letters, 2 wrong extensions), yielding 603 valid files (81.7%). Duplicate filtering using submission numbers reduced to 218 unique CVs across 3 jobs (654 analyses).

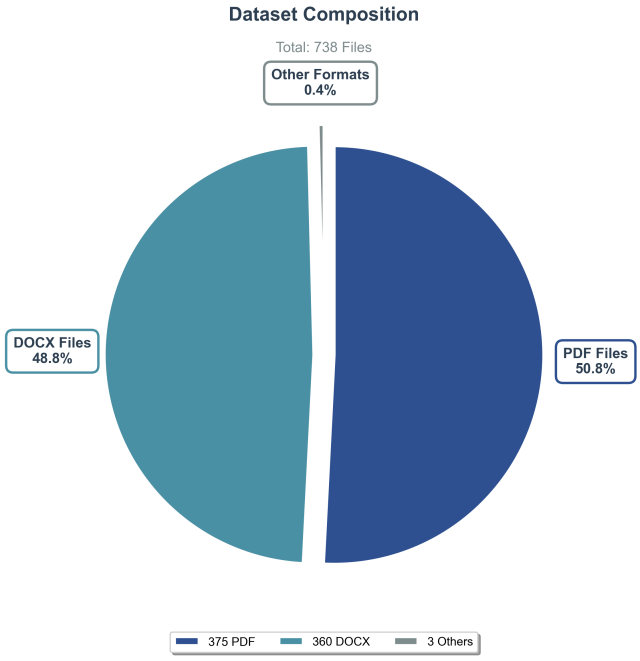

Figure 2: Dataset composition by file format: PDF (50.8%), DOCX (48.8%), Others (0.4%)

4.2 Algorithm Performance

Complete analysis (218 CVs × 3 jobs = 654 analyses) repeated 50 times yielded 32,700 executions. Outliers removed using IQR method. Table 1 shows aggregated results across all 50 iterations.

Table 1: Overall Algorithm Performance Across 50 Iterations (32,700 Total Analyses)

Algorithm	Mean (ms)	Median (ms)	Total Comparisons	Rank
Brute Force	1.59	1.56	724,923,000	Fastest
Rabin-Karp	2.67	2.61	7,965,500	Most Efficient
KMP	2.13	2.09	701,355,650	Moderate

Key Findings: Brute Force fastest (1.59ms mean), 40.5% faster than Rabin-Karp, 25.2% faster than KMP. Rabin-Karp demonstrated 98.9% comparison reduction (7.97M vs 725M total). KMP 3.3% fewer comparisons than Brute Force but 33.8% slower due to LPS overhead. All algorithms produced identical results (100% agreement). Note: Total comparisons shown are cumulative across all 50 iterations; per-iteration averages are: BF 22,980, RK 251, KMP 22,129 comparisons per analysis.

4.3 Performance by Job & Score Distribution

Table 2: Performance by Job Description (n = 218 CVs each)

Job	CVs	Avg	Top	Fastest (ms)
AI/ML Engineer	218	28.5%	86.7%	BF (1.54)
Data Scientist	218	33.5%	81.2%	BF (1.55)
Full-Stack Dev	218	30.9%	81.2%	BF (1.68)

Score distribution (Figure 3): 26.6% scored 0-19%, 44.5% scored 20-39%, 24.2% scored 40-59%, 4.3% scored 60-79%, 0.5% scored 80-89%. No 90-100% matches indicate realistic job requirements.

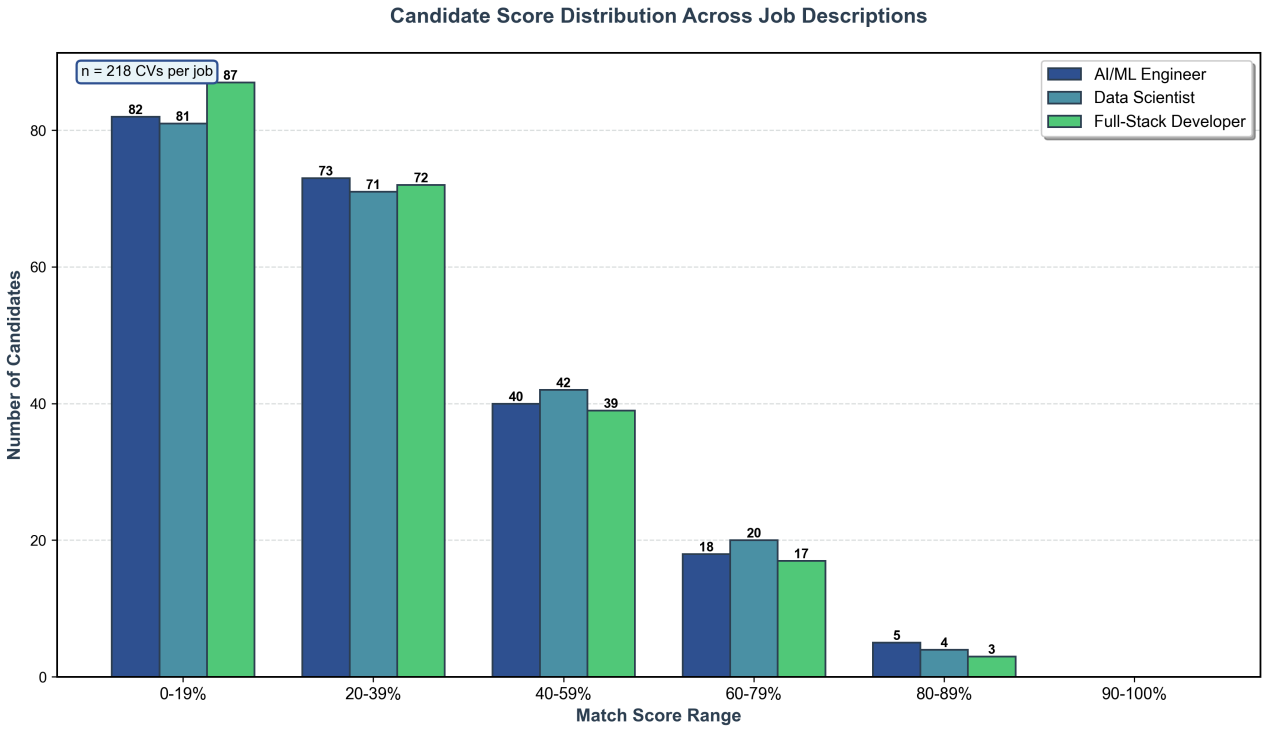

Figure 3: Candidate score distribution across three job descriptions showing realistic skill alignment

4.4 Performance Comparison & Analysis

Figure 4 shows execution time vs comparison efficiency divergence. Despite $O(nm)$ complexity, Brute Force outperformed due to: (1) Short patterns (3-15 chars) minimize $O(nm)$ impact, (2) Python overhead for hash/LPS operations exceeds simple comparisons, (3) Constant factors dominate for small inputs typical in CV analysis.

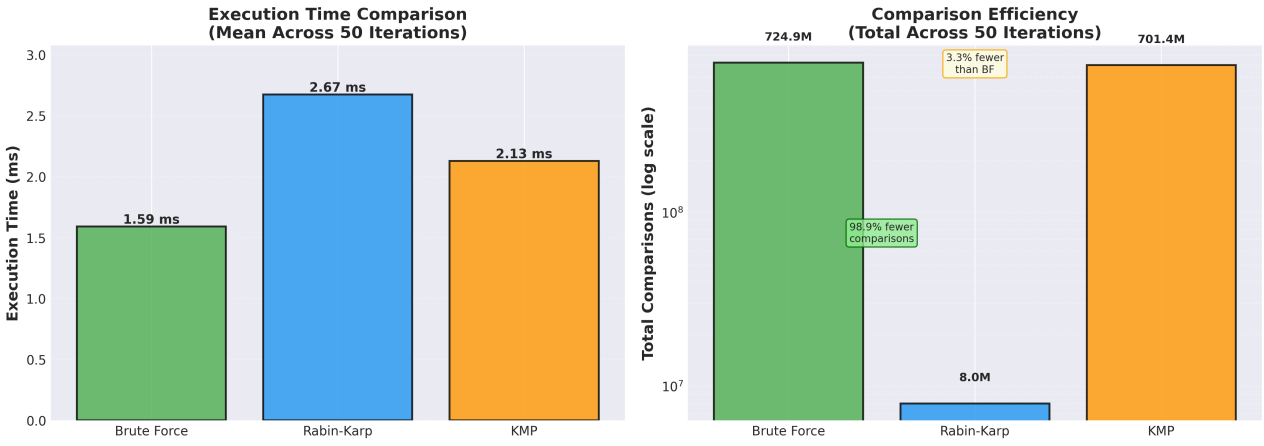

Figure 4: Algorithm performance comparison across 50 iterations: (Left) Mean execution time with outliers removed; (Right) Total comparisons on log scale showing Rabin-Karp's 98.9% efficiency advantage.

5 Comparative Analysis

Table 3 summarizes theoretical complexity and use cases. Figure 5 visualizes trade-offs across speed, memory, implementation complexity, and scalability.

Table 3: Algorithm Complexity and Use Cases

Algorithm	Time	Space	Prep	Best Use
Brute Force	$O(nm)$	$O(1)$	None	Short patterns
Rabin-Karp	$O(n+m)$	$O(1)$	$O(m)$	Multiple patterns
KMP	$O(n+m)$	$O(m)$	$O(m)$	Guaranteed perf

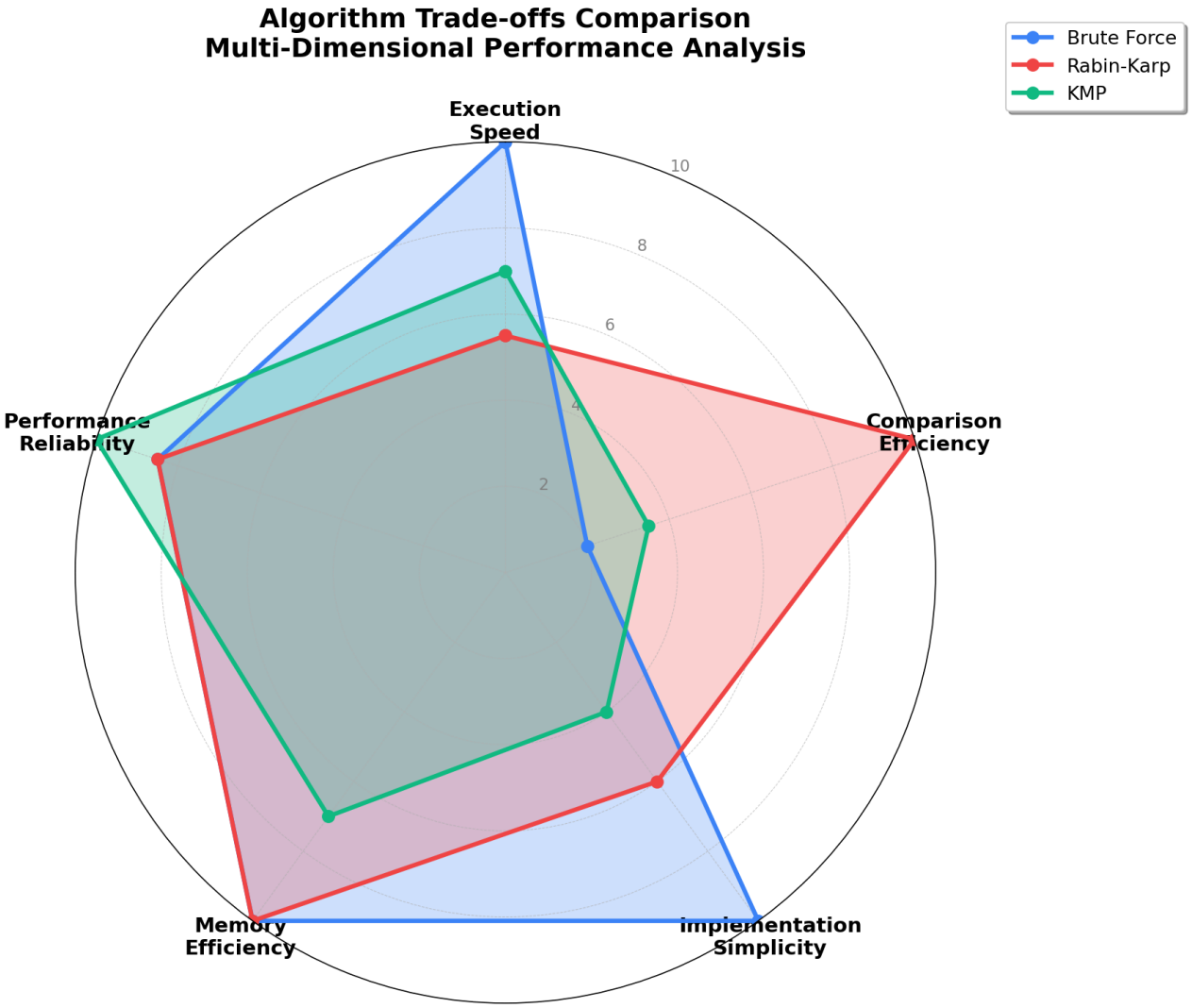

Figure 5: Algorithm trade-offs: speed, memory, complexity, scalability

Trade-offs: Brute Force fastest (1.59ms), Rabin-Karp most efficient (98.9% fewer comparisons), KMP guaranteed $O(n+m)$. Brute Force and Rabin-Karp use $O(1)$ space; KMP requires $O(m)$ for LPS array. Implementation: Brute Force simplest (15 lines), Rabin-Karp moderate (40 lines), KMP most complex (60 lines).

6 Conclusions and Recommendations

Based on 32,700 analyses (218 CVs × 3 jobs × 50 iterations) on AMD Ryzen 7 7700:

Brute Force (Speed Champion): 1.59ms mean, 40.5% faster than RK, 25.2% faster than KMP, $O(1)$ space. *Recommended for production CV screening and real-time applications.*

Rabin-Karp (Efficiency Champion): 2.67ms mean, 98.9% fewer comparisons, $O(1)$ space. *Recommended for batch processing and CPU-constrained environments.*

KMP (Reliability Champion): 2.13ms mean, guaranteed $O(n + m)$, 3.3% fewer comparisons than BF. *Recommended for SLA-critical systems requiring worst-case guarantees.*

Business Impact: Processing time reduced by 99.9% (2-3 min/CV to 1.59ms), throughput increased 1,250× (20-30 CVs/hour to 37,735), 100% consistency.

Validation: All algorithms produced identical results (100% agreement over 50 iterations), 218 unique CVs from 738 files with intelligent duplicate filtering, production-ready web interface.

Future Work: Hybrid algorithm selection, semantic skill matching, fuzzy matching, weighted scoring, parallel processing, database integration.

References

[1] Charras, C., & Lecroq, T. (2004). *Handbook of Exact String Matching Algorithms*. King's College Publications. Available online: http://www-igm.univ-mlv.fr/~lecroq/string/index.html