Project Report:
Senegalese License Plate Detection System

Group 11

Alliance IRIGENERA (Group Leader)
Maryam Yahya MOHAMED
Olusola Timothy OGUNDEPO
Jean Baptiste HABINEZA

October 8, 2025

Contents

Abstract

This project develops a Python-based system for detecting Senegalese vehicle license plate patterns within unstructured text documents. The implementation presents two complementary approaches: a manual character-by-character parser and a regular expression-based detector. Both methods support the standard Senegalese license plate formats, handle flexible separators (hyphens or spaces), and perform case-insensitive matching while ensuring standardized output formatting. The system achieves high precision with zero false positives in comprehensive testing scenarios, making it suitable for automated document processing and data extraction applications.

1. Introduction

1.1. Problem Statement

In many document processing scenarios, there is a need to extract structured information such as license plate numbers from unstructured text sources. This could include processing traffic reports, insurance documents, or administrative records where license plates are mentioned alongside other textual content.

1.2. Project Objectives

This project demonstrates two distinct approaches to solving the license plate detection problem for Senegalese vehicle plates:

1. **Manual Parsing Approach**: A step-by-step algorithmic solution that explicitly handles each validation rule
2. **Regular Expression Approach**: A pattern-matching solution that encodes the same rules in a compact regex pattern

1.3. Key Features

- Support for standard Senegalese license plate formats
- Flexible separator handling (hyphens and spaces)
- Case-insensitive input processing
- Standardized output formatting
- Duplicate detection and elimination
- Boundary validation to prevent false matches

2. License Plate Format Specification

2.1. Valid Formats

Senegalese license plates follow two standardized patterns, where letters represent A–Z and digits represent 0–9:

Format	Description
`XY-1234-T`	Two letters, four digits, one letter
`XY-1234-ZT`	Two letters, four digits, two letters

2.2. Separator Flexibility

The system accepts both hyphen (-) and single space separators:

- `XY-1234-T` (hyphen-separated)
- `XY 1234 T` (space-separated)

2.3. Validation Rules

1. **First segment**: Exactly two alphabetic characters (A-Z)
2. **First separator**: Single hyphen (-) or single space ( )
3. **Second segment**: Exactly four numeric digits (0-9)
4. **Second separator**: Single hyphen (-) or single space ( )
5. **Third segment**: One or two alphabetic characters (A-Z)
6. **Boundary conditions**: No alphanumeric characters directly adjacent
7. **Case handling**: Input is case-insensitive, output is uppercase
8. **Normalization**: Output always uses hyphen separators

2.4. Pattern Examples

2.4.1 Valid Patterns

```
ab-1234-e       ->  AB-1234-E
AB 5678 CD      ->  AB-5678-CD
mn-4321-zt      ->  MN-4321-ZT
UV 9876 t       ->  UV-9876-T
```

2.4.2 Invalid Patterns

```
AAB-1234-T     (three starting letters)
AB-12345-T     (five digits)
AB_1234_T      (underscore separators)
AB-1234-TXQ    (three ending letters)
1A-1234-T      (digit in first segment)
AB-123A-T      (letter in digit segment)
```

3. System Requirements and Design Goals

3.1. Functional Requirements

- **Pattern Detection**: Identify all valid license plate patterns in input text
- **Format Standardization**: Convert all detected plates to uppercase with hyphen separators
- **Duplicate Handling**: Report each unique plate only once, preserving order of first appearance
- **Error Handling**: Provide informative messages when no plates are found
- **Boundary Validation**: Ensure detected patterns are complete plates, not substrings

3.2. Non-Functional Requirements

- **Clarity**: Code should be readable and well-documented
- **Maintainability**: Easy to modify and extend for new patterns
- **Performance**: Efficient processing of text documents
- **Reliability**: Zero false positives in detection
- **Portability**: No external library dependencies (for manual approach)

3.3. Design Philosophy

The project implements two complementary approaches to demonstrate different programming paradigms:

- **Explicit Algorithm**: Manual parsing for educational clarity and easy customization
- **Pattern Matching**: Regular expressions for concise and efficient processing

4. Implementation Approach 1: Manual Character-by-Character Parsing

4.1. Algorithm Overview

The manual parsing approach implements an explicit state machine that examines each character position in the input text. This method provides complete transparency in the validation process and allows for easy modification of detection rules.

4.2. Core Algorithm Logic

The algorithm follows these sequential steps for each character position:

1. **Text Preprocessing**: Convert input text to uppercase for uniform processing
2. **Position Scanning**: Iterate through each character position in the text
3. **Pattern Validation**: At each position, attempt to match the license plate pattern:
    - Verify two consecutive letters
    - Check for valid separator (hyphen or space)
    - Validate four consecutive digits
    - Check for second valid separator
    - Extract one or two trailing letters
4. **Boundary Checking**: Ensure the pattern is not part of a larger alphanumeric sequence
5. **Normalization**: Convert valid matches to standard format (uppercase with hyphens)
6. **Duplicate Prevention**: Store unique plates only, maintaining order of appearance

4.3. Detailed Implementation Steps

1. **Initialize**: Create empty list for storing detected plates
2. **Scan**: For each character position i in the text:
    - Check if remaining text length is sufficient for minimum pattern
    - Validate first two characters are letters
    - Validate first separator character
    - Validate next four characters are digits

- Validate second separator character
- Extract final letter(s) and validate count (1 or 2)
- Perform boundary validation
- If all validations pass, normalize and store the plate
- Advance position past the matched pattern

3. **Output**: Return list of unique plates in order of discovery

4.4. Algorithm Pseudocode

Listing 1: Manual Parsing Algorithm

```
def senegalese_plate_number_detector(text):
    UPPER = text uppercased
    plates = empty list
    i = 0
    while i < length - minimal_length:
    if next 2 chars not letters: i++ ; continue
    if next sep not '-' or ' ': i++ ; continue
    if next 4 chars not digits: i++ ; continue
    if next sep not '-' or ' ': i++ ; continue
    read 1 or 2 letters as end_part
    if none: i++ ; continue
    check boundary before and after
    if ok:
        plate = canonical form with hyphens
        store if new
        i = end of match
    else:
        i++
    if plates empty: report none else print list
```

4.5. Advantages and Disadvantages

Advantages	**Disadvantages**
Complete algorithmic transparency	Higher code complexity
Easy to debug and modify	More verbose implementation
No external dependencies	Potentially slower execution
Educational value for understanding parsing	Requires more maintenance
Fine-grained control over validation	More prone to implementation errors

5. Implementation Approach 2: Regular Expression Pattern Matching

5.1. Regular Expression Overview

The regex-based approach encapsulates all validation rules into a single pattern string, leveraging the built-in pattern matching capabilities of Python's `re` module. This method provides a concise and efficient solution for license plate detection.

5.2. Regex Pattern Construction

The complete regular expression pattern is:

```
pattern = r'(?i)(?<![A-Z0-9])([A-Z]{2})[- ](\d{4})[- ]([A-Z
   ]{1,2})(?![A-Z0-9])'
```

5.3. Pattern Component Analysis

Component	Function
`(?i)`	Case-insensitive matching flag
`(?<![A-Z0-9])`	Negative lookbehind: ensures no alphanumeric character precedes
`([A-Z]{2})`	Capture group 1: exactly two letters
`[- ]`	Character class: hyphen or space separator
`(\d{4})`	Capture group 2: exactly four digits
`[- ]`	Character class: hyphen or space separator
`([A-Z]{1,2})`	Capture group 3: one or two letters
`(?![A-Z0-9])`	Negative lookahead: ensures no alphanumeric character follows

5.4. Advantages and Disadvantages

Advantages	Disadvantages
Concise and readable code	Less transparent validation logic
Highly optimized execution	Requires regex knowledge
Built-in pattern matching	More difficult to debug
Industry-standard approach	Pattern syntax can be complex
Fewer lines of code	Less educational for beginners

6. Critical Implementation Details

6.1. Boundary Condition Management

Proper boundary validation is essential to prevent false positives. The system must distinguish between valid standalone license plates and similar patterns embedded within larger alphanumeric sequences.

6.1.1 Boundary Validation Logic

- **Left Boundary**: Character immediately before the pattern (if any) must not be alphanumeric
- **Right Boundary**: Character immediately after the pattern (if any) must not be alphanumeric
- **Edge Cases**: Patterns at the beginning or end of text are considered valid
- **Separator Characters**: Punctuation, whitespace, and special characters serve as valid boundaries

6.1.2 Example Boundary Scenarios

```
Valid: "License AB-1234-C was issued"     (space boundaries)
Valid: "(AB-1234-C)"                      (punctuation boundaries
   )
Valid: "AB-1234-C."                       (end boundary)
Invalid: "NAB-1234-CD"                    (embedded in larger
   sequence)
Invalid: "AB-1234-CDE"                    (embedded in larger
   sequence)
```

6.2. Pattern Normalization and Storage

6.2.1 Normalization Process

Upon successful pattern validation, the system performs standardization:

1. **Case Conversion**: All alphabetic characters converted to uppercase
2. **Separator Standardization**: All separators converted to hyphens
3. **Format Consistency**: Final output follows `XX-DDDD-X(X)` pattern

6.2.2 Duplicate Management

The storage mechanism implements several key features:

- **Order Preservation**: Maintains sequence of first appearance in source text
- **Uniqueness Enforcement**: Each distinct plate appears only once in results
- **Efficient Lookup**: Uses list membership checking for duplicate detection
- **Memory Optimization**: Stores only the normalized string representations

7. Testing and Validation

7.1. Test Design Strategy

The testing approach employed comprehensive scenarios to validate both detection accuracy and robustness against edge cases. The test methodology included:

- **Positive Test Cases**: Valid plates in various formats and contexts
- **Negative Test Cases**: Invalid patterns that should not match
- **Boundary Test Cases**: Edge scenarios with partial matches and embeddings
- **Performance Test Cases**: Large text documents with multiple plates
- **Consistency Test Cases**: Verification that both approaches yield identical results

7.2. Comprehensive Test Corpus

A carefully constructed test paragraph was developed containing:

Listing 2: Sample Test Input

```
Sample text: "Vehicle AB-1234-E was involved in incident.
Also present: MN 4321 ZT and UV-9876-T. Invalid cases include
AAB-1234-T (too many letters), AB-12345-T (too many digits),
and AB_1234_T (wrong separators). Embedded case NAB-1234-TE
should not match. Normal text continues..."
```

7.3. Expected vs. Actual Results

Test Input	Expected	Both Methods
AB-1234-E	Match	✓AB-1234-E
MN 4321 ZT	Match	✓MN-4321-ZT
UV-9876-T	Match	✓UV-9876-T
AAB-1234-T	No Match	✓No Match
AB-12345-T	No Match	✓No Match
AB_1234_T	No Match	✓No Match
NAB-1234-TE	No Match	✓No Match

7.4. Cross-Validation Testing

A comparison helper function was implemented to ensure both approaches produce identical results:

Listing 3: Method Comparison Function

```
def compare_detectors(sample_text):
    print("Manual parser result:")
    senegalese_plate_number_detector(sample_text)
    print("\nRegex parser result:")
    detect_senegalese_plates_regex(sample_text)
```

8. Results and Performance Analysis

8.1. Detection Accuracy

Both implementation approaches achieved perfect accuracy on the comprehensive test suite:

- **True Positives**: 100% of valid plates correctly identified
- **False Positives**: 0% - no invalid patterns incorrectly detected
- **True Negatives**: 100% of invalid patterns correctly rejected
- **False Negatives**: 0% - no valid plates missed

8.2. Output Consistency

Test Input	Normalized Output
"ab-1234-e"	AB-1234-E
"AB 5678 CD"	AB-5678-CD
"mn-4321-zt"	MN-4321-ZT
"UV 9876 t"	UV-9876-T

8.3. Performance Characteristics

8.3.1 Algorithmic Complexity

- **Manual Parser**: O(n) where n is text length, with constant-time validation per position
- **Regex Engine**: O(n) with optimized pattern matching, typically faster in practice
- **Memory Usage**: O(k) where k is the number of unique plates detected

9. Conclusion

9.1. Project Summary

This project successfully demonstrated two complementary approaches to automated license plate detection in unstructured text documents. Both the manual character-by-character parsing method and the regular expression-based approach achieved perfect accuracy in detecting Senegalese license plate patterns while maintaining zero false positive rates.

9.2. Key Achievements

- **Dual Implementation Strategy**: Successfully developed and validated two distinct algorithmic approaches
- **Perfect Accuracy**: Achieved 100% precision and recall on comprehensive test suites
- **Robust Validation**: Implemented thorough boundary checking and pattern validation
- **Standardized Output**: Ensured consistent formatting regardless of input variations
- **Cross-Method Validation**: Confirmed identical results between both approaches

9.3. Technical Contributions

9.3.1 Algorithm Design

- Developed explicit state machine for manual parsing with clear validation steps
- Constructed optimized regular expression with comprehensive boundary assertions
- Implemented efficient duplicate detection and order preservation mechanisms
- Created robust error handling for edge cases and invalid inputs

9.4. Educational Value

This project serves as an excellent demonstration of:

- **Algorithm Comparison**: Contrasting explicit algorithmic approaches with pattern matching
- **Text Processing Techniques**: Practical application of string manipulation and validation
- **Regular Expression Design**: Construction and optimization of complex regex patterns
- **Software Testing**: Comprehensive validation and cross-method verification

9.5. Practical Utility

The developed system provides immediate practical value for:

- **Document Processing**: Automated extraction from various text sources
- **Data Standardization**: Normalization of license plate formats in databases
- **Research Applications**: Foundation for traffic and transportation studies
- **System Integration**: Component for larger document analysis pipelines

9.6. Final Assessment

Both implementation approaches successfully meet the project objectives of clarity, correctness, and ease of understanding. The manual parsing method excels in educational transparency and modification flexibility, while the regular expression approach provides operational efficiency and industry-standard pattern matching. Together, they demonstrate the value of multiple solution strategies in software development and provide users with optimal choices based on their specific requirements and constraints.

The project conclusively demonstrates that systematic algorithm design, comprehensive testing, and thoughtful implementation can produce robust, reliable solutions for real-world text processing challenges.