EXPERIMENT 3:

To Implement Pattern Matching Method Used for Information Retrieval (IR)

Zoya Momin

Department of Computer Engineering

M.H Saboo Siddik College of Engineering

Mumbai, India

zoya.221257.co@mhssce.ac.in

I. INTRODUCTION

Information Retrieval (IR) refers to the process of searching, locating, and retrieving relevant information from large collections of unstructured text data. A fundamental component of IR is **pattern matching**, which involves scanning text to find occurrences of a specific keyword or phrase.Pattern matching can be implemented using various methods, from simple brute-force approaches to advanced algorithms like **Knuth-Morris-Pratt (KMP)**, **Boyer-Moore**, or **Regular Expressions**.In this experiment, we focus on implementing a basic pattern-matching technique in Python, demonstrating how preprocessing (case normalization, punctuation removal) and search algorithms work together to locate patterns efficiently in textual data.

Steps to Implement Pattern Matching for Information Retrieval

Objective : To implement a basic pattern-matching technique that scans a given text or document to locate a specific keyword or phrase—mimicking a fundamental Information Retrieval (IR) operation.

1. **Define the Problem** : Identify the text corpus and the pattern (keyword/phrase) to search for.
2. **Select the Text Corpus**: Choose the input text (paragraph, article, or document) where the pattern will be searched.
3. **Preprocess the Text**: Convert text to lowercase. Remove punctuation and special characters for uniformity.
4. **Implement Brute-Force Pattern Matching**: Check every substring of the document for a match with the pattern.
5. **Implement Regex Matching (Optional)**: Use Python's re.finditer() for efficient and flexible searching.
6. **Test the Implementation**: Run the code with different input documents and patterns to verify correctness.
7. **View the Output**: Display positions where the pattern occurs in the document.
8. **Draw the Conclusion**: Compare brute-force and regex approaches, noting their efficiency and applicability.

Code :

```
import re

# Step 1: Preprocessing Function
def preprocess(text):
    text = text.lower()  # Convert to lowercase
    text = re.sub(r'[^\w\s]', '', text)  # Remove punctuation
    return text

# Step 2: Brute-Force Pattern Matching
def simple_pattern_matching(pattern, document):
    matches = []
    pattern_length = len(pattern)
    document_length = len(document)
    for i in range(document_length - pattern_length + 1):
        if document[i:i + pattern_length] == pattern:
            matches.append(i)
    return matches

# Step 3: Regex Matching
def regex_match(pattern, document):
    return [m.start() for m in re.finditer(pattern, document)]

# Sample Text and Pattern
# document_text = "Information Retrieval is a core concept in data science."
# pattern_text = "information retrieval"
document_text = "i like waffle but i like pancake more"
pattern_text = "waffle"

# Preprocess both
processed_doc = preprocess(document_text)
processed_pattern = preprocess(pattern_text)

# Brute-force search
brute_matches = simple_pattern_matching(processed_pattern, processed_doc)

# Regex search
regex_matches = regex_match(processed_pattern, processed_doc)

# Output
print("Brute-force matches found at positions:", brute_matches)
print("Regex match positions:", regex_matches)
```

Output:

```
Brute-force matches found at positions: [7]
Regex match positions: [7]
```

II. CONCLUSION

Pattern matching is foundational to IR systems. While brute-force is easy to understand and implement, it's inefficient for large data. Advanced techniques and libraries make large-scale retrieval practical.Pattern matching is a core technique in Information Retrieval systems. Brute-force pattern matching is simple to implement and works well for small datasets, but it becomes inefficient for large-scale data due to its **O(m × n)** time complexity.Using **advanced algorithms** like KMP or tools like **Regex** improves efficiency and flexibility, especially for real-time searches or large document collections. For enterprise-level solutions, indexing and search engines such as **Elasticsearch** or **Apache Lucene** provide scalability and high performance.