Set-Theoretic IR Models

Assignment Report #05

Submitted To:

Dr. Syed Khaldoon Khurshid

Submitted by:

Muhammad Yaqoob

Reg. No.:

2021-CS-118

Department of Computer Science

University of Engineering and Technology (UET) Lahore

Table Of Content

List Of Images

Introduction

The Boolean Extended Information Retrieval (IR) Model represents an advanced approach to document retrieval that leverages Boolean algebra principles to process and match user queries against document collections. Unlike traditional information retrieval methods, this model extends beyond simple binary matching by incorporating complex logical operations and providing more sophisticated query processing capabilities.
Regular search uses simple words like "AND," "OR," and "NOT" to find things. The extended Boolean model is like using superpowers with these words. We can be more specific. For example, if you want to find “Fruits OR Vegetables BUT NOT Processed Food," it can do that easily.

Objective

The primary objectives of the Boolean Extended IR Model include:

- Develop a robust document retrieval mechanism using Boolean logic
- Enable complex query processing through AND, OR, and NOT operations
- Create an efficient document representation and matching system
- Provide a flexible framework for information retrieval
- Implement a scalable search mechanism without relying on external libraries

Scenario

Imagine you're looking for articles about sports, but you want to focus on "soccer" OR "basketball" WITHOUT anything related to "golf."
Boolean Query: Using a classic Boolean search, you might enter: "soccer OR basketball NOT golf." While this helps, you might still encounter some articles that include golf indirectly.
Extended Boolean Model Approach: This approach refines the search even further:

- "soccer" OR "basketball": This part works the same, retrieving articles about soccer or basketball.
- WITHOUT "golf": The Extended Boolean Model filters out anything related to golf, ensuring you only get articles about soccer or basketball without unrelated golf content.

In essence, the Extended Boolean Model provides better precision, allowing you to fine-tune your search results by clearly defining what should be included and excluded. It's like handing the search engine an exact checklist of your preferences!

Mathematical Expression

The general framework for the **Extended Boolean Model** can be expressed mathematically:

- Let A, B, C, etc., represent sets of documents or terms.
- Logical operators allow for their combination:

    - **OR**: Represents the union of sets.
    - **AND**: Represents the intersection of sets.
    - **NOT**: Represents the complement of a set.
- Queries can involve more complex combinations, such as: (A OR B) AND ~ C) This example retrieves documents that contain either A or B but excludes any that include C.

Using this approach, you can construct precise queries to include desired terms, exclude irrelevant ones, and combine them in flexible ways, enhancing the relevance of retrieved documents.

Steps to Implement the Model

- Step 1: Document Preprocessing
- Step 2: Query Processing Architecture
- Step 3: Mathematical Representation
- Step 4: Boolean Algebraic Operations

Step 1: Document Preprocessing

- Tokenize input documents
- Generate a unique vocabulary set

```
def preprocess_documents(self) -> None:
        """
        Preprocess documents into vocabulary and processed documents
        """
        for doc in self.documents:
            # Tokenize and clean terms
            terms = self._clean_and_tokenize(doc)
            self.vocab.update(terms)
            self.processed_docs.append(terms)

        # Convert vocab to sorted list for consistent indexing
        self.vocab = sorted(list(self.vocab))

        # Create term document matrix
        self.term_doc_matrix = self.create_term_document_matrix()

        # Initialize term weights (optional)
        self._initialize_term_weights()

def _clean_and_tokenize(self, document: str) -> List[str]:
        """
        Clean and tokenize a document

        Args:
            document (str): Input document string

        Returns:
            List[str]: Cleaned and tokenized terms
        """
        # Basic cleaning: lowercase, split
        return document.lower().split()

def _initialize_term_weights(self) -> None:
    """
    Initialize term weights based on term frequency
    """
    for term in self.vocab:
        # Simple weight calculation based on document frequency
        doc_frequency = sum(self.get_term_vector(term))
        self.term_weights[term] = 1 + (doc_frequency /
len(self.documents))

def get_term_vector(self, term: str) -> List[int]:
    """
    Get binary vector for a given term

    Args:
        term (str): Term to retrieve vector for

    Returns:
        List[int]: Binary vector representing term presence
    """
    if term not in self.vocab:
        # logger.warning(f"Term '{term}' not in vocabulary")
        return [0] * len(self.documents)

    index = self.vocab.index(term)
    return [doc[index] for doc in self.term_doc_matrix]
```

- Create a binary term-document matrix

```
def create_term_document_matrix(self) -> List[List[int]]:
    """
    Create binary term-document matrix

    Returns:
        List[List[int]]: Binary matrix representing term presence
    """
    matrix = []
    for doc in self.processed_docs:
        vector = [1 if term in doc else 0 for term in self.vocab]
        matrix.append(vector)
    return matrix
```

- Normalize and clean text tokens

```
# Create term-document matrix
def create_term_document_matrix(vocab, processed_docs):
    matrix = []
    for doc in processed_docs:
        vector = [1 if term in doc else 0 for term in vocab]
        matrix.append(vector)

    return matrix

term_doc_matrix = create_term_document_matrix(vocab, processed_docs)
```

Step 2: Query Processing Architecture

- Parse input query
- Identify Boolean operators
- Apply logical operations sequentially
- Generate result vector

```
def process_query(self, query: str) -> List[int]:
    """
    Process complex Boolean query

    Args:
        query (str): Boolean query string

    Returns:
        List[int]: Result vector indicating matching documents
    """
    # logger.info(f"Processing query: {query}")

    terms = query.split()
    result_vector = []

    i = 0
    while i < len(terms):
        term = terms[i]

        if term == "AND":
            next_term = terms[i + 1]
            result_vector = self.and_operation(result_vector, self.get_term_vector(next_term))
            i += 2
        elif term == "OR":
            next_term = terms[i + 1]
            result_vector = self.or_operation(result_vector, self.get_term_vector(next_term))
            i += 2
        elif term == "NOT":
            next_term = terms[i + 1]
            result_vector = self.not_operation(self.get_term_vector(next_term))
            i += 2
        else:
            result_vector = self.get_term_vector(term)
            i += 1

    return result_vector
```

- Rank matching documents

```
def rank_documents(self, result_vector: List[int]) -> List[Tuple[int,
str]]:
    """
    Rank documents based on result vector

    Args:
        result_vector (List[int]): Binary vector of matching documents

    Returns:
        List[Tuple[int, str]]: List of ranked documents with indices
    """
    ranked_docs = [
        (i, self.documents[i])
        for i in range(len(result_vector))
        if result_vector[i] == 1
    ]
    return ranked_docs
```

Step 3: Mathematical Representation

Term-Document Matrix Representation

Let V = {t1, t2, ..., tn} be the vocabulary set

Let D = {d1, d2, ..., dm} be the document collection

Term-Document Matrix A is defined as an m × n binary matrix:

A[i,j] = { 1, if term tj exists in document di 0, otherwise }

Boolean Algebraic Operations

AND Operation

$\wedge(x, y) = \min(x, y)$

- Returns 1 only if both terms are present
- Represents intersection of term sets

OR Operation

$\vee(x, y) = \max(x, y)$

- Returns 1 if either term is present
- Represents union of term sets

NOT Operation

$\neg(x) = 1 - x$

- Inverts the binary representation
- Excludes documents containing specific terms

```
def and_operation(self, vec1: List[int], vec2: List[int]) -> List[int]:
    """
    Perform AND operation on two vectors

    Args:
        vec1 (List[int]): First binary vector
        vec2 (List[int]): Second binary vector

    Returns:
        List[int]: Resulting binary vector
    """
    return [x & y for x, y in zip(vec1, vec2)]

def or_operation(self, vec1: List[int], vec2: List[int]) -> List[int]:
    """
    Perform OR operation on two vectors

    Args:
        vec1 (List[int]): First binary vector
        vec2 (List[int]): Second binary vector

    Returns:
        List[int]: Resulting binary vector
    """
    return [x | y for x, y in zip(vec1, vec2)]

def not_operation(self, vec: List[int]) -> List[int]:
    """
    Perform NOT operation on a vector

    Args:
        vec (List[int]): Input binary vector

    Returns:
        List[int]: Inverted binary vector
    """
    return [1 - x for x in vec]
```

Example

Lets understand the Boolean extended model with an example

Sample Documents

1. "algorithm optimization sorting"
2. "algorithm dynamic graph theory"
3. "optimization dynamic programming"

Vocabulary Generation

{algorithm, optimization, sorting, dynamic, graph, theory, programming}

Term-Document Matrix

[1 1 1 0 0 0 0]
[1 0 0 1 1 1 0]
[0 1 0 1 0 0 1]

Query Processing Examples

Query: "algorithm AND optimization"

- Retrieves documents containing both terms

Query: "algorithm OR dynamic"

- Retrieves documents with either term

Query: "algorithm NOT dynamic"

- Excludes documents with "dynamic"

Data Flow Diagram

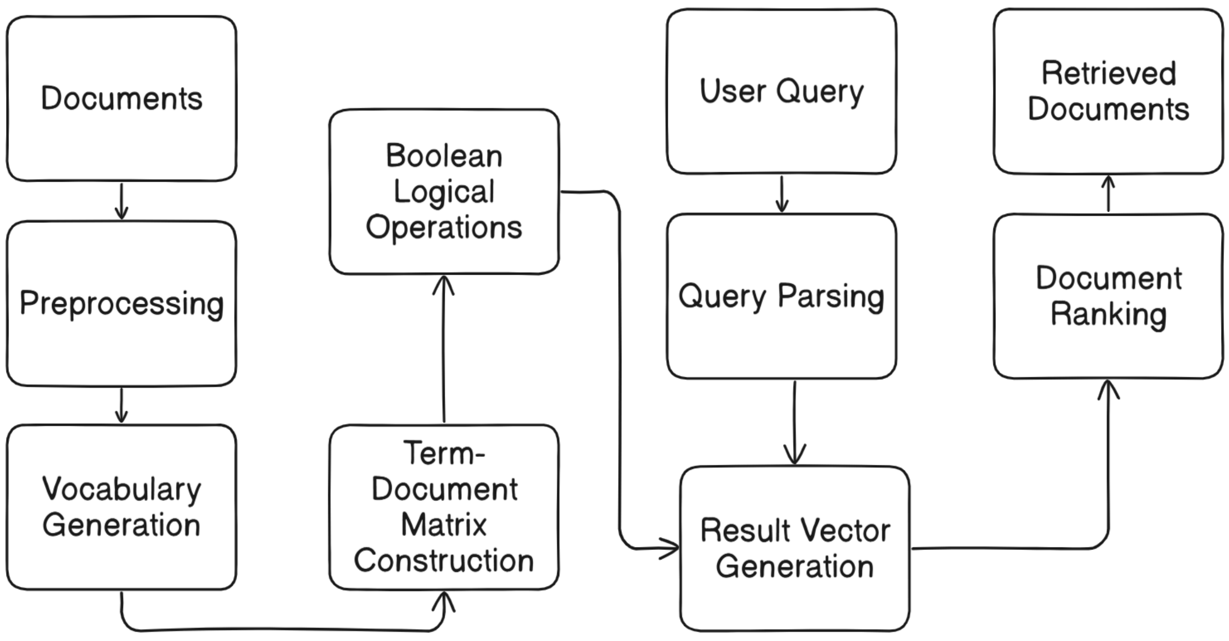

Figure 1. Data Flow Diagram

Performance Characteristics

Computational Complexity

Time Complexity: O(n * m)

- n: number of terms
- m: number of documents

Space Complexity: O(n * m)

Strengths

- Simple implementation
- Fast query processing
- Flexible logical operations
- No external library dependencies

Limitations

- Binary representation lacks semantic nuance
- No inherent relevance ranking

- Performance degrades with large document collections

Conclusion

The Boolean Extended IR Model provides a fundamental approach to document retrieval using logical operations. By implementing complex Boolean algebra principles, we create a versatile information retrieval system that can handle sophisticated query requirements.