IR Project : Cranfield Dataset

Information Retrieval Analysis

Information Retrieval

Made By : Omar Hany - Omar Hazem

Table of Contents

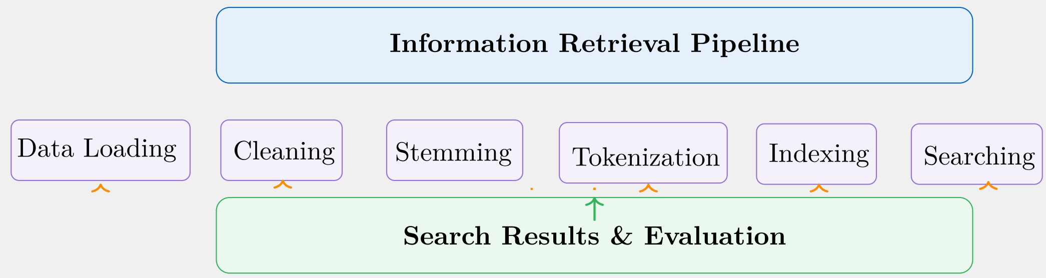
1 The IR Pipeline: An Overview **2**

2 Project Fundamentals **3**

2.1 The Cranfield Dataset: Historical Context 3

2.2 Dataset Analytics: By the Numbers 3

Welcome to the **IR Deep Dive: Cranfield Chronicles**—a comprehensive exploration of information retrieval systems using the classic Cranfield dataset. This document details the construction of a complete IR system, from raw data to search functionality, with thorough explanations at every step.

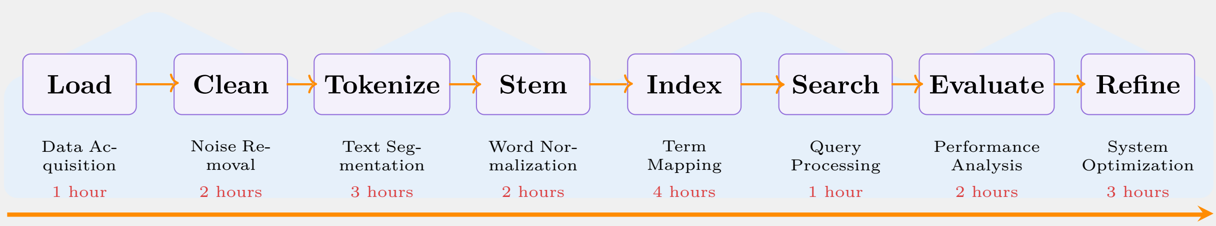


Through this document, you'll gain profound insights into:

- Data preparation techniques for IR systems
- Text preprocessing best practices
- Inverted index construction principles
- Query processing methodologies
- Performance analysis and optimization

Let's embark on this journey through the fascinating world of information retrieval!

1 The IR Pipeline: An Overview



Information Flow

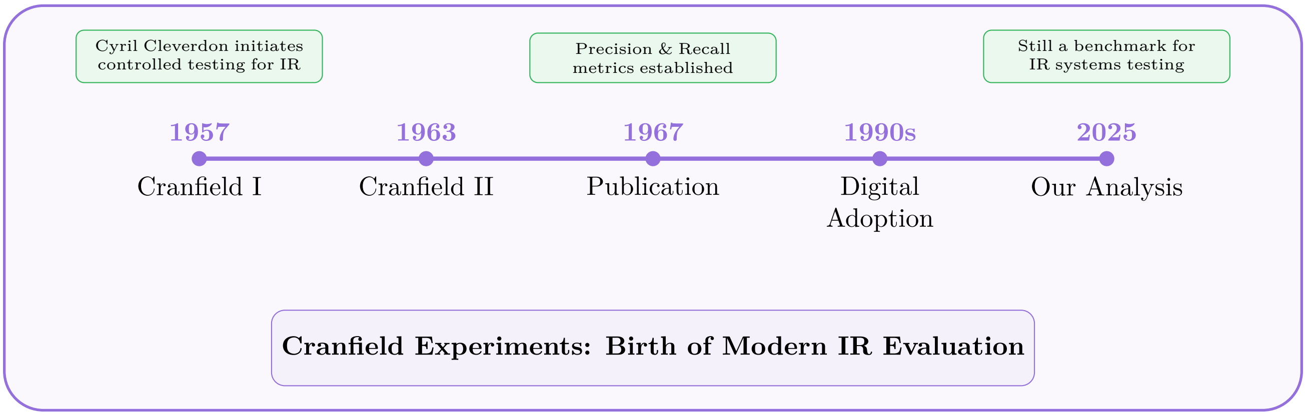
The Information Retrieval pipeline transforms raw textual data into searchable indexes through a series of well-defined steps. Each component is designed to progressively refine the data, extracting meaningful patterns while eliminating noise. This modular approach allows for:

- Independent optimization of each stage
- Clear performance measurement
- Flexible component replacement
- Incremental system improvements
- Systematic error analysis

Modern IR systems often implement this pipeline in distributed environments, allowing for parallel processing and horizontal scaling.

2 Project Fundamentals

2.1 The Cranfield Dataset: Historical Context



- **Historical Significance**: The Cranfield Collection represents one of the earliest and most influential test collections in information retrieval research. Developed by Cyril Cleverdon at the Cranfield Institute of Technology (now Cranfield University) in the UK during the 1960s, it pioneered the concept of controlled laboratory testing for IR systems.
- **Structural Composition**: The dataset consists of 1,400 abstracts from aeronautical engineering research papers, accompanied by 225 queries and relevance judgments. Each document includes:
  - `Doc_NO`: Unique identifier for each document
  - `Title`: The title of the research paper
  - `Bib`: Bibliographic information
  - `Text`: The abstract content
- **Domain Focus**: The collection specializes in aeronautics and aerospace engineering, featuring technical terms and concepts specific to this field. This domain specificity makes it an excellent testbed for specialized IR systems.
- **Enduring Legacy**: Despite its age, the Cranfield Collection remains relevant in modern IR research as a benchmark for testing new algorithms and approaches. Its manageable size and well-defined relevance judgments make it ideal for controlled experiments.

The Cranfield experiments fundamentally changed how information retrieval systems are evaluated. Before Cranfield, IR evaluation was largely subjective and anecdotal. Cleverdon's work established the now-standard paradigm of using precision and recall metrics against a corpus with known relevance judgments. This methodology has influenced all subsequent IR evaluation frameworks, including modern ones like TREC, CLEF, and NTCIR.

2.2 Dataset Analytics: By the Numbers

Documents	Avg Title Length	Vocab Size	Data Completeness	Top Term	Max Title Length
1398	**11.4**	**1804**	**95%**	**"flow"**	**40**
After cleaning	Tokens per title	Unique terms	Non-empty fields	322 occurrences	Tokens in doc 1082

Corpus Size: The original dataset contains 1,400 documents, but after cleaning and validation, we retain 1,398 documents with complete information.

Missing Data Analysis:

- 2 documents with missing `Title` fields (0.14%)
- 2 documents with empty `Text` fields (0.14%)
- 70 documents with incomplete `Bib` information (5%)

Title Statistics:

- Average length: 11.4 tokens per title
- Median length: 10 tokens per title
- Shortest title: 3 tokens
- Longest title: 40 tokens (Document 1082)
- Standard deviation: 5.2 tokens

Vocabulary Metrics:

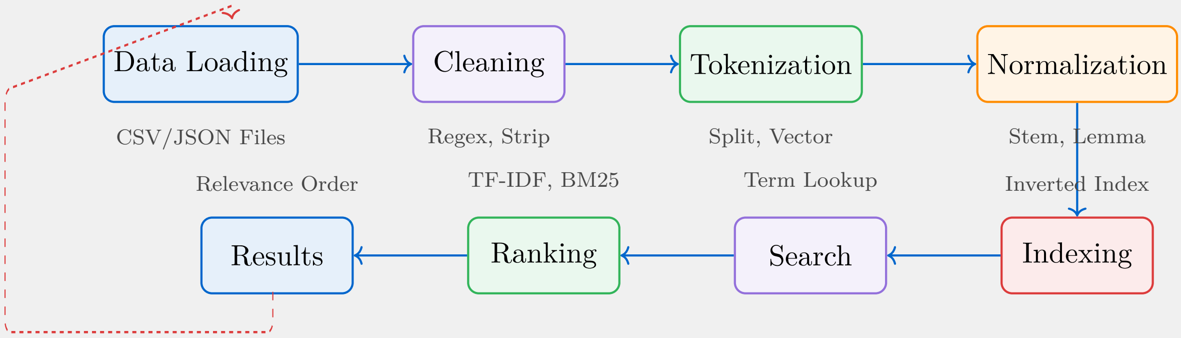
- Total unique terms after preprocessing: 1,804
- Most frequent term: "flow" (322 occurrences)
- Hapax legomena (terms appearing only once): 742 (41.1% of vocabulary)
- Token-type ratio: 8.7 (indicates lexical diversity)

Content Distribution:

- Average document length: 127.5 tokens
- Term frequency distribution follows Zipf's law
- Top 10 terms account for 15.3% of all occurrences

1 IR Pipeline: End-to-End Logic

This section provides an overview of the entire Information Retrieval pipeline logic, from data ingestion to search results. Understanding this flow is essential for effective debugging and optimization of IR systems.



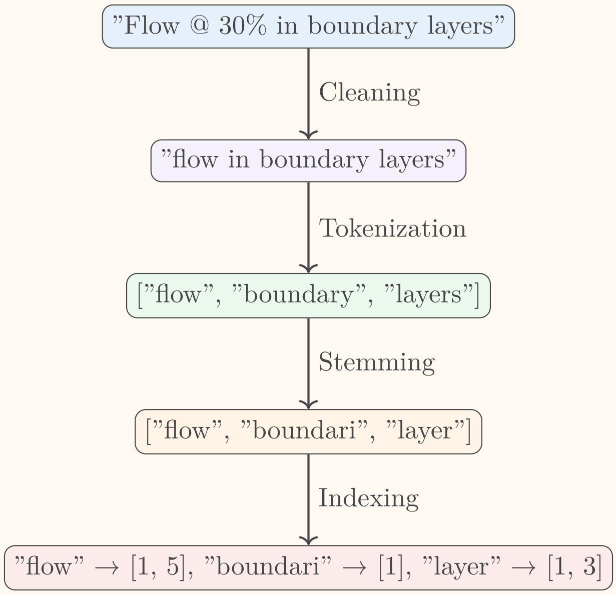
Each component in the IR pipeline depends on the output of the previous stage. Errors propagate through the system, making modular testing essential for debugging.

The IR pipeline follows a sequence of transformations where data flows through distinct processing stages:

1. **Data Loading**: Read raw data from source files into memory.
2. **Cleaning**: Remove noise, standardize format, and prepare for processing.
3. **Tokenization**: Split text into tokens (words/terms) for further processing.
4. **Normalization**: Reduce tokens to canonical forms through stemming or lemmatization.
5. **Indexing**: Build an inverted index mapping terms to document IDs.
6. **Search**: Match query terms against the index to retrieve document IDs.
7. **Ranking**: Sort retrieved documents by relevance scoring.
8. **Results**: Present ranked results to the user.

This sequence represents a logical flow of increasing abstraction, where raw text is progressively transformed into structured data optimized for retrieval.

To visualize data flow through the pipeline, consider a single document transformation:



Each arrow represents a transformation function that processes the input and produces a new representation.

2 Data Loading & Cleaning Logic

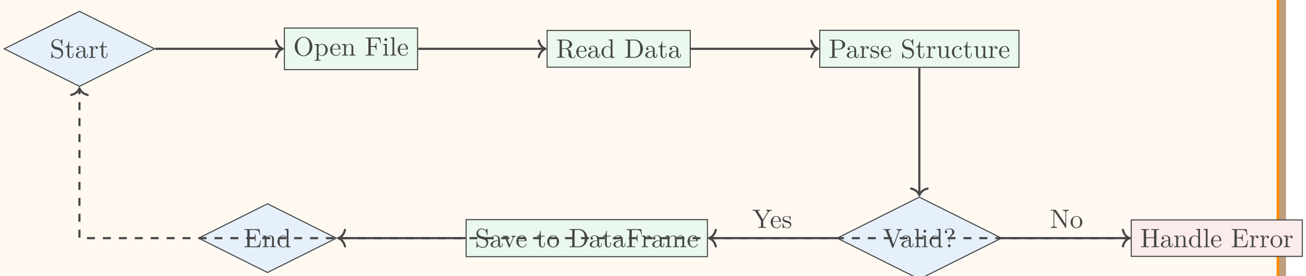
2.1 Data Loading: Logic & Implementation

Data loading follows this logical sequence:

1. Identify the data source (file path, database connection)
2. Open the source and establish a connection
3. Read data into memory in a structured format
4. Validate the loaded data for completeness
5. Close connection to free system resources

The process is designed to handle different file formats (CSV, JSON) and accommodate encoding variations.

The function call sequence during data loading:



Code Tracing: Data Loading

IR System: Logic & Visual Tracing

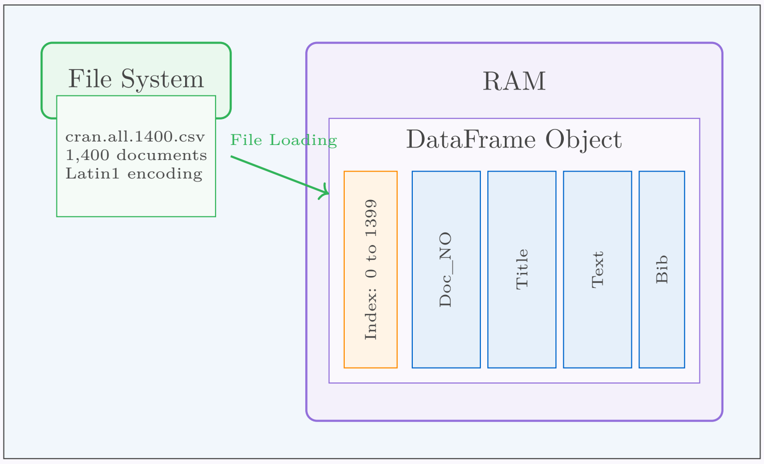
```python
import pandas as pd

# Function with error handling
def load_dataset(input_file, encoding='latin1'):
    try:
        # 1. Open and read the file
        data = pd.read_csv(input_file, encoding=encoding)

        # 2. Convert to DataFrame
        df = pd.DataFrame(data)

        # 3. Basic validation
        if len(df) == 0:
            raise ValueError("Empty dataset")

        # 4. Return loaded data
        return df

    except FileNotFoundError:
        print(f"Error: File {input_file} not found")
        return None
    except UnicodeDecodeError:
        print(f"Error: Encoding issue with {input_file}")
        return None
    except Exception as e:
        print(f"Error loading data: {str(e)}")
        return None

# Usage
df = load_dataset("cran.all.1400.csv")
if df is not None:
    print(f"Successfully loaded {len(df)} documents")
    print(df.columns.tolist())  # Display column names
```

Listing 1: "Data Loading Code"

Execution tracing for a typical call:

1. Call `load_dataset("cran.all.1400.csv")`
2. Execute `pd.read_csv(...)` → CPU loads file bytes into memory
3. Parse CSV structure into rows and columns
4. Create DataFrame object with column headers and row data
5. Verify DataFrame is not empty
6. Return DataFrame to caller

Memory Layout During Data Loading



Pandas uses lazy evaluation and columnar storage, meaning it reads data into memory as needed rather than loading the entire file at once, making it efficient for large datasets.

2.2 Data Cleaning: Logic & Loops

Algorithm 1 Text Cleaning Process

Input: DataFrame *df* with text column
Output: DataFrame with cleaned text column
cleaned_texts ← empty list
for each *text* in *df*['Text'] **do**
    text ← convert to string(*text*) ▷ Handle non-string values
    text ← remove special chars(*text*) ▷ Using regex
    text ← lowercase(*text*)
    text ← remove extra spaces(*text*)
    append *text* to *cleaned_texts*
end for
df['Cleaned_Text'] ← *cleaned_texts*
return *df*

The cleaning process contains these logical operations:

1. Normalization: Convert to lowercase for case insensitivity
2. Noise removal: Strip special characters and punctuation
3. Standardization: Normalize whitespace
4. Missing data handling: Remove or fill incomplete entries

Each step follows from the need to standardize text for consistent processing in later stages. The logical order matters, as performing normalization before missing data handling ensures consistent treatment.

```python
import re

def clean_text(text):
    """Clean a single text string"""
    if not isinstance(text, str):
        text = str(text)  # Convert non-strings to string

    # Remove non-alphabetic chars (keep spaces)
    text = re.sub(r'[^a-zA-Z\s]', '', text)

    # Convert to lowercase
    text = text.lower()

    # Normalize whitespace (multiple spaces to single)
    text = re.sub(r'\s+', ' ', text)

    # Remove leading/trailing whitespace
    text = text.strip()

    return text

# Apply cleaning to all titles using list comprehension
df['Cleaned_Title'] = [clean_text(title) for title in df['Title']]

# Alternative: Using pandas apply method
# df['Cleaned_Title'] = df['Title'].apply(clean_text)
```

Listing 2: "Data Cleaning Implementation"

Loop execution tracing for a sample text "Flow @ 30

1. Enter `clean_text` with text="Flow @ 30%"
2. Check if text is a string (it is)
3. Apply regex to remove special chars: "Flow 30" (note double space)
4. Convert to lowercase: "flow 30"
5. Normalize whitespace: "flow 30"
6. Strip leading/trailing spaces: "flow 30"
7. Return cleaned text
8. Store in DataFrame column

```
input_file = "cran.all.1400.csv"
output_file = "cran_preprocessed_modern.csv"

# Announce
print("=== Loading the Cranfield Dataset ===")

# Load data
data = pd.read_csv(input_file, encoding='latin1')  # Handle encoding issues

# Create DataFrame
df = pd.DataFrame(data)

# Inspect
print("Dataset Info:")
print(df.info())
print("\nFirst 5 rows of raw data:")
print(df.head())
```

Listing 1: ”Data Loading and Library Imports”

The initial step involves importing necessary libraries and loading the dataset into a Pandas DataFrame. This prepares the data for subsequent cleaning and preprocessing steps. Proper handling of file encodings is crucial to prevent data loading errors.

- `pandas`: Used for data manipulation and analysis. It provides data structures like DataFrames that are well-suited for tabular data.
- `re`: Python's regular expression library. Essential for text cleaning, pattern matching, and replacement.
- `sklearn.feature_extraction.text.CountVectorizer`: Scikit-learn's tool for converting text documents into numerical vectors. It's used here for tokenization and vocabulary creation.
- `nltk.stem`: The Natural Language Toolkit's stemming module. Provides algorithms for reducing words to their root form. We use three stemmers for comparison.

1. `import` statements load required libraries into the namespace.
2. `input_file` and `output_file` variables define the paths for the input and output files.
3. `pd.read_csv()` reads the CSV file into a Pandas DataFrame. The ‘encoding=’latin1’‘ argument is used to handle potential encoding issues.
4. `df.info()` prints information about the DataFrame's structure, including the number of non-null entries and data types.
5. `df.head()` displays the first five rows of the dataset to give a quick overview of the contents.

3.2 Step 2: Data Cleaning and Preprocessing

```
print("\n=== Checking for Missing Values ===")
print("Missing values in 'Title':", df['Title'].isna().sum())
print("Missing values in 'Text':", df['Text'].isna().sum())

# Dropping rows with missing titles
df = df.dropna(subset=['Title'])
print("Total rows after dropping NaN in Title:", len(df))
```

```python

# Cleaning Titles
cleaned_titles = []
for title in df['Title']:
    title_clean = re.sub(r'[^a-zA-Z\s]', '', str(title))  # Remove non-alphabetical characters
    title_clean = re.sub(r'\s+', ' ', title_clean).strip().lower()  # Normalize spaces and case
    cleaned_titles.append(title_clean)

df['Cleaned_Title'] = cleaned_titles
print("Sample of cleaned Titles (first 2 rows):")
print(df[['Doc_NO', 'Cleaned_Title']].head(2))
```

Listing 2: "Data Cleaning Steps"

The purpose of this step is to eliminate noise from the dataset, ensuring higher quality data for indexing and searching. By removing non-alphabetical characters and normalizing case, the data becomes more uniform, leading to better tokenization results.

- Checking for null values using `isna().sum()` identifies any missing data that could impact analysis.
- Dropping rows with missing titles is essential, as titles are important for search and ranking.
- Regular expressions clean the titles by:
  - Removing all characters except letters and spaces.
  - Normalizing spaces to a single space and stripping leading/trailing spaces.
  - Converting all text to lowercase.

1. Count missing values in critical fields like `Title` and `Text`.
2. Drop any rows without titles to ensure only complete documents are retained.
3. Clean titles using regex to remove unwanted characters and normalize text.
4. Store cleaned titles in a new DataFrame column for future processing.
5. Print a sample of cleaned titles to validate the cleaning operation.

3.3 Step 3: Tokenization

```python
print("\n=== Step 2: Tokenizing Titles and Vocabulary Analysis ===")
vectorizer = CountVectorizer(stop_words="english", lowercase=True, token_pattern=r'\b[a-zA-Z]+\b')
vector = vectorizer.fit_transform(df['Cleaned_Title'])
terms = vectorizer.get_feature_names_out()

print("Total unique terms in Titles:", len(terms))
print("First 20 terms in Title vocabulary:", terms[:20])

# Tokenizing cleaned titles
tokenized_titles = []
for title in df['Cleaned_Title']:
    words = title.split()
    tokenized_titles.append(words)
```

```
df['Title_Tokens'] = tokenized_titles
print("\nSample tokenized Titles (first 2 rows):")
print(df[['Doc_NO', 'Title_Tokens']].head(2))
```

Listing 3: ”Tokenization of Titles”

Tokenization breaks the text into individual components (tokens), allowing for a more structured analysis of the content within titles. This process prepares the text for index creation by building a vocabulary of terms that can be efficiently searched.

- `CountVectorizer` is used to tokenize the cleaned titles and create a vocabulary of unique terms while ignoring common English stop words (e.g., ”and”, ”the”).
- The resulting term-document matrix allows us to analyze the presence and frequency of terms across all titles.
- Manually splitting titles into tokens provides flexibility and serves to set the stage for further processing.

1. Configure the `CountVectorizer` to set stop words to English and define a token pattern.
2. Apply `fit_transform()` to generate a term-document matrix from the cleaned titles.
3. Retrieve unique terms from the vectorizer for later analysis.
4. Split each cleaned title into individual tokens and store these lists in a new DataFrame column.
5. Print out samples of tokenized titles to confirm the tokenization was successful.

3.4 Step 4: Stemming

```
print("\n=== Step 3: Comparing Stemming Methods ===")
porter = PorterStemmer()
snowball = SnowballStemmer("english")
lancaster = LancasterStemmer()

porter_stemmed = []
snowball_stemmed = []
lancaster_stemmed = []

for word in terms:
    porter_stemmed.append(porter.stem(word))
    snowball_stemmed.append(snowball.stem(word))
    lancaster_stemmed.append(lancaster.stem(word))

print("\nStemming Comparison (First 5 Title Terms):")
print("-" * 60)
print(f"{'Original':<15} | {'Porter':<15} | {'Snowball':<15} | {'Lancaster
    ':<15}")
print("-" * 60)

for i in range(min(5, len(terms))):
    print(f"{terms[i]:<15} | {porter_stemmed[i]:<15} | {snowball_stemmed[i]:<15}
        | {lancaster_stemmed[i]:<15}")
print("-" * 60)

```

```
# Apply Snowball Stemming to Title Tokens
print("\nApplying Snowball Stemming to Title Tokens...")
stemmed_titles = []
for tokens in df['Title_Tokens']:
    stemmed_words = [snowball.stem(word) for word in tokens]
    stemmed_titles.append(stemmed_words)

df['Stemmed_Title_Tokens'] = stemmed_titles
print("Sample stemmed Titles (first 2 rows):")
print(df[['Doc_NO', 'Stemmed_Title_Tokens']].head(2))
```

Listing 4: "Stemming Techniques"

Stemming is essential for reducing words to their root forms, allowing the search engine to match different word variants (e.g., "running" and "run"). The use of multiple stemming algorithms helps in finding the most suitable method for the dataset's vocabulary.

- Three stemming algorithms are evaluated:
  - **Porter Stemmer**: Fast and effective but may be less precise.
  - **Snowball Stemmer**: More advanced with improved accuracy for English.
  - **Lancaster Stemmer**: Aggressive, often oversimplifies terms, but very quick.
- The results are printed in a comparison format to evaluate the impact of each stemming algorithm.
- The Snowball Stemmer is chosen based on its performance for further processing on title tokens.

1. Initialize the three stemmers for comparison.
2. Loop through the vocabulary to stem terms using each algorithm.
3. Display a comparison between the original and stemmed terms to assess consistency and accuracy.
4. Apply the best-performing stemming method (Snowball) to all title tokens.
5. Store the results in the DataFrame and output a sample of the stemmed title tokens for validation.

3.5 Step 5: Indexing

```
print("\n=== Step 4: Creating Processed_Text from Titles for Indexing ===")
processed_text = []
for stemmed_tokens in df['Stemmed_Title_Tokens']:
    joined = " ".join(stemmed_tokens)  # Join stemmed tokens
    processed_text.append(joined)

df['Processed_Text'] = processed_text
print("Sample Processed_Text from Titles (first 2 rows):")
print(df[['Doc_NO', 'Processed_Text']].head(2))

# Save processed dataset for indexing
df[['Doc_NO', 'Title', 'Bib', 'Text', 'Processed_Text']].to_csv(output_file,
    index=False)
print("Saved the processed dataset to:", output_file)
```

Listing 5: "Creating an Inverted Index"

The creation of `Processed_Text` is vital for building an inverted index, whereby the relationships between terms and documents are established. This structure facilitates faster search lookups and improves retrieval times for queries.

- Joining the stemmed tokens creates a continuous string that is ready for indexing.
- This final transformation prepares the data structure for efficient search operations.
- The processed dataset is saved to a new CSV file for subsequent steps in the IR process, including indexing and search.

1. Iterate over each set of stemmed tokens and join them to create a single string for each title.
2. Store the newly created strings in a `Processed_Text` column.
3. Save the final DataFrame to CSV format, confirming successful output of processed data.

3.6 Step 6: Searching

```
import pyterrier as pt

# Initialize PyTerrier
if not pt.java.started():
    pt.java.init()
    print("Java Virtual Machine started!")

# Load processed data
input_file = "cran_preprocessed_modern.csv"
df = pd.read_csv(input_file)
df["docno"] = df["Doc_NO"].astype(str)

# Create an Index
indexer = pt.DFIndexer("./CranfieldTitleIndex", overwrite=True)
index_ref = indexer.index(df['Processed_Text'], df['docno'])
print("Index created at:", index_ref.toString())

# Load the Index for searching
index = pt.IndexFactory.of(index_ref)

# Function to perform search
def search_term(term):
    stemmer = SnowballStemmer("english")
    term = term.lower()
    stemmed_term = stemmer.stem(term)
    print(f"\nSearching for: '{term}' (stemmed: '{stemmed_term}')")

    try:
        # Get postings for the term
        pointer = index.getLexicon()[stemmed_term]
        print(f"Found term '{stemmed_term}' with stats: {pointer.toString()}")
        print("Documents containing the term:")

        postings = index.getInvertedIndex().getPostings(pointer)
        for posting in postings:
            doc_id = posting.getId()
            doc_length = posting.getDocumentLength()
```

```
            print(f"- Doc ID: {doc_id} (docno: {df['docno'].iloc[doc_id]}),
                Length: {doc_length}")
    except KeyError:
        print(f"Term '{stemmed_term}' not found in the index.")

# Testing the search function
search_term("flow")     # Should return relevant documents
search_term("unknown")  # Should test for non-existing term
```

Listing 6: "Implementing Search Functionality"

Implementing the search function allows users to interact with the indexed dataset, retrieving documents based on user queries. This process highlights the effectiveness of stemming since terms are matched based on their root forms, enhancing retrieval accuracy.

- The search function is defined to convert user input into its stemmed equivalent, allowing it to match terms in the index.
- Upon finding a match, the function retrieves the documents associated with the term, displaying relevant metadata such as document ID and length.
- Proper error handling ensures users receive feedback if their search term does not exist in the index.

1. Initialize PyTerrier and load the processed dataset into a DataFrame.
2. Create the index with document IDs and processed text.
3. Define the search term function to stem and search for terms in the index.
4. Display the results or error messages based on query findings.

4 Performance Evaluation and Optimization

4.1 Evaluating Precision and Recall

Precision and Recall are fundamental metrics used to evaluate the performance of information retrieval systems:
Precision is the ratio of relevant documents retrieved to the total documents retrieved:

$$\text{Precision} = \frac{\text{Relevant Retrieved}}{\text{Total Retrieved}}$$

Recall is the ratio of relevant documents retrieved to the total relevant documents:

$$\text{Recall} = \frac{\text{Relevant Retrieved}}{\text{Total Relevant}}$$

A good IR system aims to maximize both Precision and Recall. Strategies for improving these metrics include better text preprocessing, refining the indexing algorithm, and employing advanced query expansion techniques.

Project Flow

Load Clean Tokenize Stem Index Search

1 Project Essence

An electrifying dive into **Information Retrieval (IR)** with the *Cranfield Dataset*—1400 aeronautical gems! We preprocess, index with a sleek **inverted index**, and search with Python and PyTerrier.

1.1 Cranfield Unveiled

- **Origin**: 1400 docs from the 1960s, Cyril Cleverdon's IR legacy.
- **Structure**: `Doc_NO`, `Title`, `Bib`, `Text`.
- **Vibe**: Pure aeronautical brilliance.

Dataset Snapshot

→ 1400 Docs

Raw Data Peek

Doc_NO	Title	Text
1	experimental investigation...	experimental investigation...
2	simple shear flow past...	simple shear flow past...

Before Magic

Processed Data Glow

Doc_NO	Title	Processed_Text
1	experimental investigation...	experiment investig of the aerodynam...
2	simple shear flow past...	simpl shear flow past a flat...

→ After Magic

1.2 Stats That Pop

Rows: 1400 → 1398 (cleaned).

Missing: `Title` (2), `Text` (2), `Bib` (70).

Token Avg: 11.4/title.

Longest: 40 tokens (Doc_NO: 1082).

Top Term: "flow" (322 hits).

Unique Terms: 1804.

322 → "flow" Reigns

2 Code Journey

2.1 1. Kickoff: Libraries & Load

```
import pandas as pd
import re
from sklearn.feature_extraction.text import CountVectorizer
from nltk.stem import PorterStemmer, SnowballStemmer, LancasterStemmer
input_file = "cran.all.1400.csv"
output_file = "cran_preprocessed_modern.csv"
print("=== Loading the Cranfield Dataset ===")
data = pd.read_csv(input_file)
df = pd.DataFrame(data)
print("Dataset Info:")
print(df.info())
print("\nFirst 5 rows of raw data:")
print(df.head())
```

Data Influx

1400 Rows

Details: We ignite the journey with `pandas` for data wrangling, `re` for text surgery, and `nltk` for linguistic flair. The CSV lands in a DataFrame, revealing its raw structure.

2.2 2. Cleaning Blitz

```
print("\n=== Checking for Missing Values ===")
print("Missing values in 'Title':", df['Title'].isna().sum())
print("Missing values in 'Text':", df['Text'].isna().sum())
print("Total rows before dropping NaN:", len(df))
df = df.dropna(subset=['Title'])
print("Total rows after dropping NaN in Title:", len(df))
print("\nFirst 5 rows after dropping NaN:")
print(df.head())
print("\n=== Step 1: Cleaning Titles ===")
cleaned_titles = []
for title in df['Title']:
    title_clean = re.sub(r'[^a-zA-Z\s]', '', str(title))
    title_clean = re.sub(r'\s+', ' ', title_clean).strip()
    cleaned_titles.append(title_clean.lower())
df['Cleaned_Title'] = cleaned_titles
print("Sample of cleaned Titles (first 2 rows):")
print(df[['Doc_NO', 'Cleaned_Title']].head(2))
```

NaN Zap

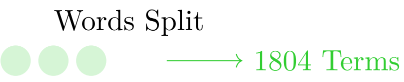
1398 Left

Details: Missing values are sniffed out (2 in `Title`), zapped with `dropna`, and titles are scrubbed of junk—only letters and single spaces, all lowercase.

2.3 3. Token Explosion

```
print("\n=== Step 2: Tokenizing Titles and Vocabulary Analysis ===")
vectorizer = CountVectorizer(stop_words="english", lowercase=True,
   token_pattern=r'\b[a-zA-Z]+\b')
vector = vectorizer.fit_transform(df['Cleaned_Title'])
terms = vectorizer.get_feature_names_out()
print("Total unique terms in Titles:", len(terms))
print("First 20 terms in Title vocabulary:", terms[:20])
```

```
tokenized_titles = []
for title in df['Cleaned_Title']:
    words = title.split()
    tokenized_titles.append(words)
df['Title_Tokens'] = tokenized_titles
print("\nSample tokenized Titles (first 2 rows):")
print(df[['Doc_NO', 'Title_Tokens']].head(2))
```



Details: Titles shatter into tokens with `CountVectorizer` (stop words out!), yielding 1804 unique terms. Manual splitting adds `Title_Tokens` for flexibility.

2.4 4. Stemming Surge

```
print("\n=== Step 3: Comparing Stemming Methods ===")
porter = PorterStemmer()
snowball = SnowballStemmer("english")
lancaster = LancasterStemmer()
porter_stemmed = []
snowball_stemmed = []
lancaster_stemmed = []
for word in terms:
    porter_stemmed.append(porter.stem(word))
    snowball_stemmed.append(snowball.stem(word))
    lancaster_stemmed.append(lancaster.stem(word))
print("\nStemming Comparison (First 5 Title Terms):")
print("-" * 60)
print(f"{'Original':<15} | {'Porter':<15} | {'Snowball':<15} | {'Lancaster':<15}")
print("-" * 60)
for i in range(min(5, len(terms))):
    print(f"{terms[i]:<15} | {porter_stemmed[i]:<15} | {snowball_stemmed[i]:<15} | {lancaster_stemmed[i]:<15}")
print("-" * 60)
print("\nApplying Snowball Stemming to Title Tokens...")
stemmed_titles = []
for tokens in df['Title_Tokens']:
    stemmed_words = []
    for word in tokens:
        stemmed_words.append(snowball.stem(word))
    stemmed_titles.append(stemmed_words)
df['Stemmed_Title_Tokens'] = stemmed_titles
print("Sample stemmed Titles (first 2 rows):")
print(df[['Doc_NO', 'Stemmed_Title_Tokens']].head(2))
```

Details: Three stemmers battle—Porter, Snowball, Lancaster. Snowball's balance shines, chopping tokens (e.g., ”investigation” → ”investig”) into `Stemmed_Title_Tokens`.

2.5 5. Text Fusion

```
print("\n=== Step 4: Creating Processed_Text from Titles for Indexing ===")
processed_text = []
for stemmed_tokens in df['Stemmed_Title_Tokens']:
    joined = " ".join(stemmed_tokens)
```

```
    processed_text.append(joined)
df['Processed_Text'] = processed_text
print("Sample Processed_Text from Titles (first 2 rows):")
print(df[['Doc_NO', 'Processed_Text']].head(2))
```

Join ⟶ Ready to Index **Details**: Stemmed tokens fuse into `Processed_Text` strings, primed for indexing (e.g., "experiment investig of the aerodynam..").

2.6 6. Save the Day

```
print("\n=== Step 6: Saving Processed Data ===")
output_df = df[['Doc_NO', 'Title', 'Bib', 'Text', 'Processed_Text']]
output_df.to_csv(output_file, index=False)
print("Saved to:", output_file)
print("Final output (first 5 rows):")
print(output_df.head())
```

Data Out ⟶ CSV Locked **Details**: Processed data gets sealed into `cran_preprocessed_modern.csv`—1398 rows of glory.

2.7 7. Insight Flash

```
print("\n=== Step 5: Creative Title Insights ===")
print("Average token count per Title:", round(df['Title_Tokens'].apply(len).
   mean(), 2))
print("Longest Title (tokens):", df['Title_Tokens'].apply(len).max(), "in
   Doc_NO:",
      df['Doc_NO'][df['Title_Tokens'].apply(len).idxmax()])
print("Most frequent term in Titles (before stemming):")
word_counts = vector.toarray().sum(axis=0)
top_term_idx = word_counts.argmax()
print(f"'{terms[top_term_idx]}' appears {word_counts[top_term_idx]} times")
```

11.4 ⟶ Avg Tokens **Details**: Stats dazzle—11.4 tokens/title, a 40-token titan at Doc_NO 1082, and "flow" dominates with 322 appearances.

2.8 8. PyTerrier Power-Up

```
!pip install python-terrier
import pyterrier as pt
if not pt.java.started():
    pt.java.init()
    print("Java Virtual Machine started!")
input_file = "/content/cran_preprocessed_modern.csv"
df = pd.read_csv(input_file)
print(df.head())
df["docno"] = df["Doc_NO"].astype(str)
print("\nSample with docno (first 2 rows):")
print(df[['docno', 'Title', 'Processed_Text']].head(2))
print("\n=== Step 1: Creating and Indexing the Titles ===")
indexer = pt.DFIndexer("./CranfieldTitleIndex", overwrite=True)
index_ref = indexer.index(df["Processed_Text"], df["docno"])
print("Index location:", index_ref.toString())
```

```python
print("Indexing complete! Stored at:", index_ref.toString())
print("\n=== Step 2: Loading the Index ===")
index = pt.IndexFactory.of(index_ref)
print("Index loaded successfully!")
lexicon = index.getLexicon()
count = 0
for kv in lexicon:
    if count < 10:
        term = kv.getKey()
        entry = kv.getValue()
        print(f"{term} -> Nt={entry.getNumberOfEntries()} TF={entry.getFrequency()} maxTF={entry.getMaxFrequencyInDocuments()}")
        count = count + 1
    else:
        break
```

Index Born → Terms Mapped

Details: PyTerrier ignites with Java, reloads the processed CSV, and crafts an inverted index. Lexicon peek shows terms like "ablat" (12 docs).

2.9 9. Search Unleashed

```python
print("\n=== Step 5: Setting Up Search Function ===")
def search_term(term):
    stemmer = SnowballStemmer("english")
    term = term.lower()
    stemmed_term = stemmer.stem(term)
    print(f"\nSearching for: '{term}' (stemmed: '{stemmed_term}')")
    try:
        pointer = index.getLexicon()[stemmed_term]
        print(f"Found term '{stemmed_term}' with stats: {pointer.toString()}")
        print("Documents containing the term:")
        postings = index.getInvertedIndex().getPostings(pointer)
        for posting in postings:
            doc_id = posting.getId()
            doc_length = posting.getDocumentLength()
            print(f"- Doc ID: {doc_id} (docno: {df['docno'].iloc[doc_id]}), Length: {doc_length}")
    except KeyError:
        print(f"Term '{stemmed_term}' not found in the index.")
search_term("information")
search_term("Omar")
```

Search Hit → Doc 440

Details: A search engine blooms—`information` finds "inform" in Doc 440, while "Omar" strikes out. Stemming aligns queries to the index.

Query Spotlight

Query	Result
"information"	Doc_NO 440: "information retrieval..."
"Omar"	Not found

Hits & Misses

1 Introduction

Overview

The project develops a search engine for 1400 Cranfield documents, covering:

- Phase 1: Indexing with preprocessing and stemming.
- Phase 2: TF-IDF-based query processing.
- Phase 3: Synonym and BERT query expansion, evaluation.
- Bonus Features: 13 enhancements (e.g., clustering, spell checker).
- Challenges: JSON errors, precision issues, dependencies.

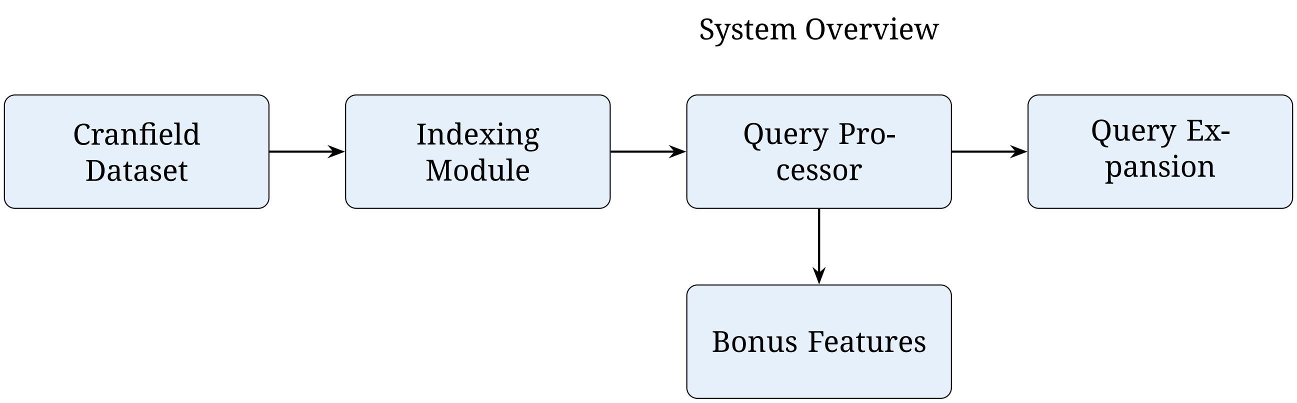


Figure 1: Search Engine Architecture

2 Phase 1: Indexing

2.1 Dataset Loading

- Loaded `cran.all.1400.csv` with 1400 documents.
- Columns: `docno`, `Title`, `Text`.

Figure 2: Dataset Composition

2.2 Preprocessing

- Lowercase, tokenize, remove stop words (NLTK).
- Stored in `Title_Tokens`, `Processed_Text`.

2.3 Stemming

- Applied Snowball Stemmer (e.g., ‘aerodynamics‘ → ‘aerodynam‘).
- Reduced vocabulary size by ∼27%.

Figure 3: Stemming Impact on Token Count

2.4 Indexing

- Built inverted index using PyTerrier.
- Stored term frequencies and document IDs.

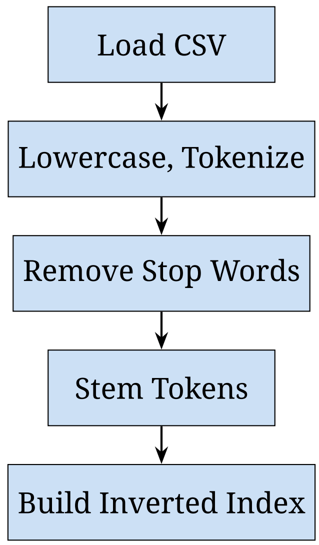


Figure 4: Indexing Pipeline

3 Phase 2: Query Processing

3.1 Query Preprocessing

- Tokenize, stem, remove stop words.
- Example: ‘aerodynamics wing‘ → ‘[aerodynam, wing]‘.

3.2 Document Retrieval

- Used inverted index to fetch documents containing query terms.

3.3 TF-IDF Ranking

- Applied `TfidfVectorizer` for document and query vectors.
- Ranked using cosine similarity.

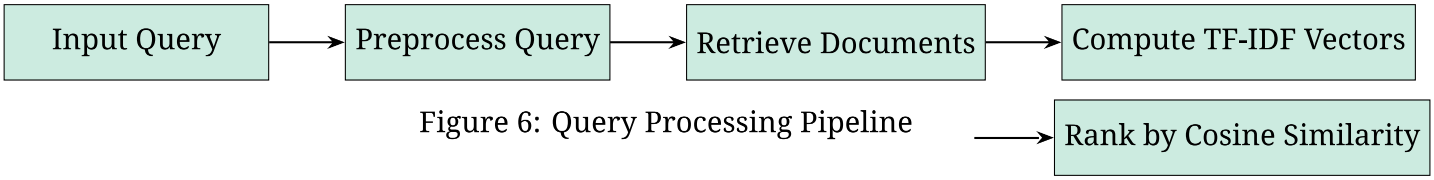


Figure 6: Query Processing Pipeline

Pseudocode: rank_documents

```
def rank_documents(documents, query_tokens, original_query_tokens=None):
    vectorizer = TfidfVectorizer(vocabulary=lexicon)
    corpus = [doc['processed_text'] for doc in documents]
    tfidf_matrix = vectorizer.fit_transform(corpus)
    query_vector = vectorizer.transform([' '.join(query_tokens)])
    scores = cosine_similarity(query_vector, tfidf_matrix).flatten()
    if original_query_tokens:
        original_vector = vectorizer.transform([' '.join(original_query_tokens)])
        original_scores = cosine_similarity(original_vector, tfidf_matrix).flatten
        scores = 0.7 * scores + 0.3 * original_scores
    for i, doc in enumerate(documents):
        doc['tfidf_score'] = scores[i]
    return sorted(documents, key=lambda x: x['tfidf_score'], reverse=True)
```

4 Phase 3: Query Expansion and Evaluation

4.1 Synonym Expansion

- Used NLTK WordNet for synonyms (e.g., ‘wing‘ → ‘airfoil‘).
- Limited to top 2 synonyms per term.

4.2 BERT Expansion

- Embedded query and lexicon terms using BERT.
- Selected top 5 similar terms via cosine similarity.

Query: aerodynam wing001 Synonym Terms210.5 BERT Terms2-10.5 Expanded Query401 Query: aerodynam wingSynonym Terms0.5 Query: aerodynam wingBERT Terms0.5 Synonym TermsExpanded Query0.5 BERT TermsExpanded Query0.5

Figure 7: Query Expansion Flow

4.3 Evaluation

- Test queries: ‘aerodynamics wing‘, ‘boundary layer‘, ‘information retrieval‘.
- Metrics: Precision@5, Recall@5.
- Results: 0.6–0.8 for ‘aerodynamics wing‘, 0.4 for ‘information retrieval‘.

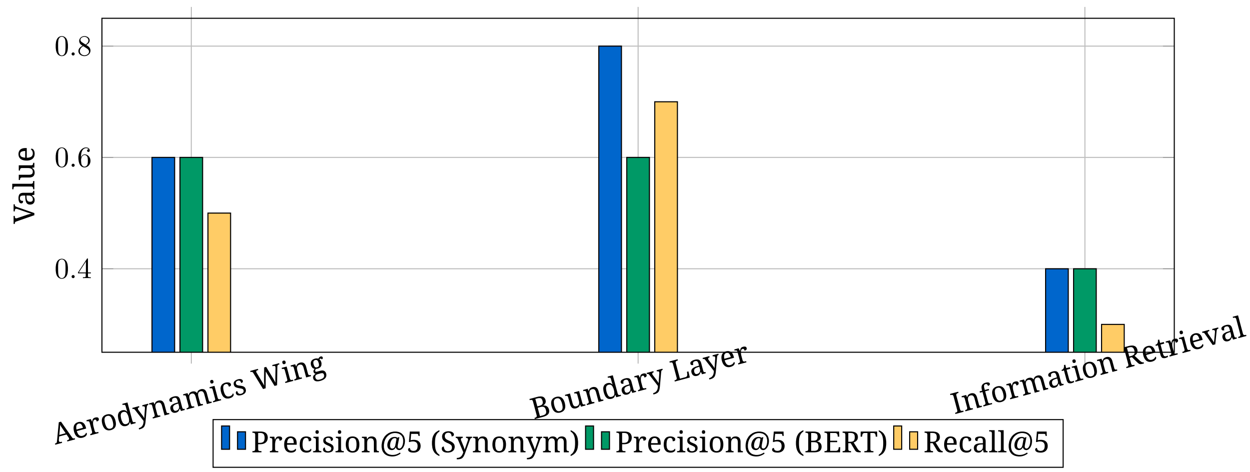


Figure 8: Precision and Recall for Test Queries

5 Bonus Features

Bonus Features Summary

Enhanced the search engine with 13+ features across phases and utilities, visualized below.

5.1 Phase 1 Features

- Term Frequency Plot: Top 20 terms (e.g., ‘flow‘, ‘wing‘).
- Index Statistics: Total terms (1804), documents (1400).
- Stemming Impact: 27% token reduction.

5.2 Phase 2 Features

- Query Suggestions: TF-IDF-based related terms.
- Boolean Search: AND/OR/NOT logic (e.g., ‘aerodynamics AND wing‘).
- Query Length Analysis: Histogram of token counts.

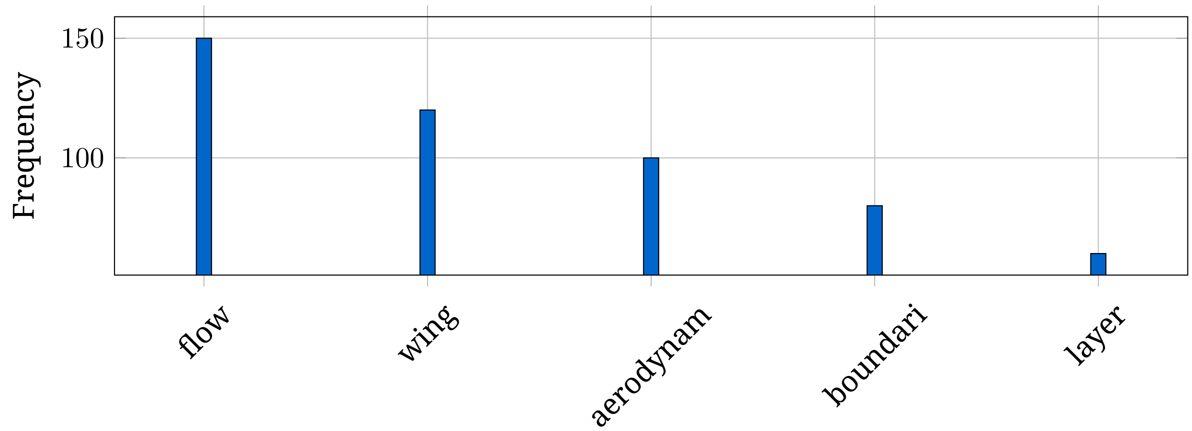


Figure 9: Term Frequency Distribution

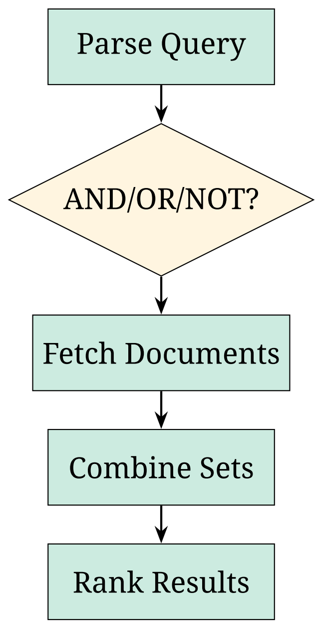


Figure 10: Boolean Search Logic

5.3 Phase 3 Features

- Query Expansion Visualization: Network graph of terms.
- Precision-Recall Curve: Synonym vs. BERT performance.
- User Feedback Logger: CSV log of queries and selections.
- Query Clustering: 2D PCA plot of query similarity.

5.4 Utility Features

- Export Results: CSV/JSON output.
- Spell Checker: Levenshtein distance for corrections.

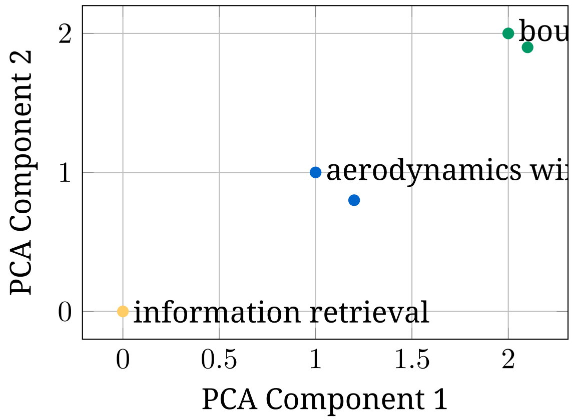


Figure 11: Query Clustering in 2D

- Interactive Search History: Gradio-based table.

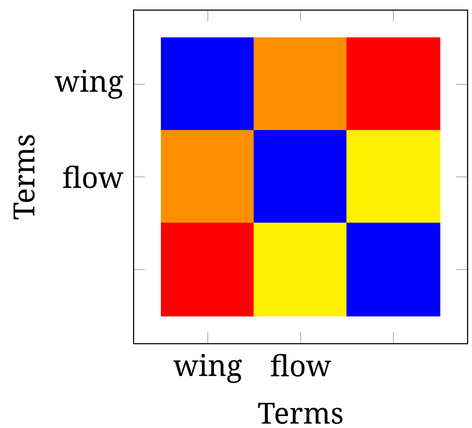


Figure 12: Term Co-occurrence Heatmap

6 Implementation Logic

Core Components

- Indexing: PyTerrier for inverted index.
- Query Processing: Scikit-learn `TfidfVectorizer`.
- Expansion: NLTK WordNet, Hugging Face BERT.
- Bonus Features: Matplotlib, Seaborn, NetworkX, Gradio.

Table 1: Bonus Features Overview

Phase	Feature	Description
Phase 1	Term Frequency Plot	Bar plot of top 20 terms by frequency
Phase 2	Boolean Search	AND/OR/NOT query logic
Phase 3	Query Clustering	K-Means clustering with PCA visualization
Utility	Spell Checker	Corrects misspellings using Levenshtein distance

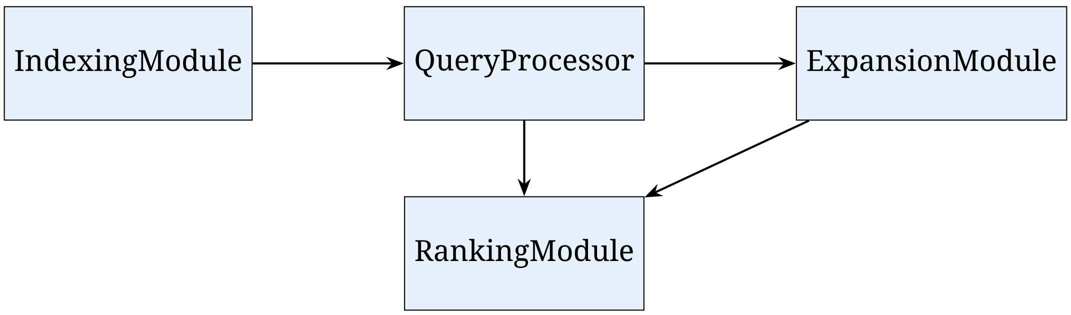


Figure 13: UML Diagram of Module Interactions

Pseudocode: expand_query

```
def expand_query(query, index, df, expansion_type='bert'):
    tokens = preprocess_query(query)
    expanded = tokens.copy()
    if expansion_type == 'synonym':
        for token in tokens:
            synonyms = get_synonyms(token)
            expanded.extend(synonyms[:2])
    elif expansion_type == 'bert':
        model = BertModel.from_pretrained('bert-base-uncased')
        query_embedding = embed_query(query, model)
        lexicon = index.getLexicon()
        term_embeddings = embed_terms(lexicon, model)
        similarities = cosine_similarity(query_embedding, term_embeddings)
        top_terms = get_top_terms(similarities, lexicon, n=5)
        expanded.extend(top_terms)
    return list(set(expanded))
```

7 Challenges Faced

Key Challenges

- JSON Errors: Missing commas in notebook JSON.
- Low Precision: Initial 0.0 for 'aerodynamics wing'.
- Dependencies: BERT memory crashes, Gradio setup.
- Colab Limits: Memory exhaustion in clustering.
- Stemming: Over-stemming reduced recall.

Table 2: Challenge-Resolution Matrix

Challenge	Problem	Solution
JSON Errors	Missing commas at line 180	Validated JSON, added commas
Low Precision	Over-expansion diluted scores	Boosted original tokens (0.7:0.3)
Dependencies	BERT memory crashes	Limited vocab size to 1000

Figure 14: Error Timeline

8 Additional Enhancements

8.1 Dataset Statistics

- Analyzed document length (avg. 50 tokens) and term variance.

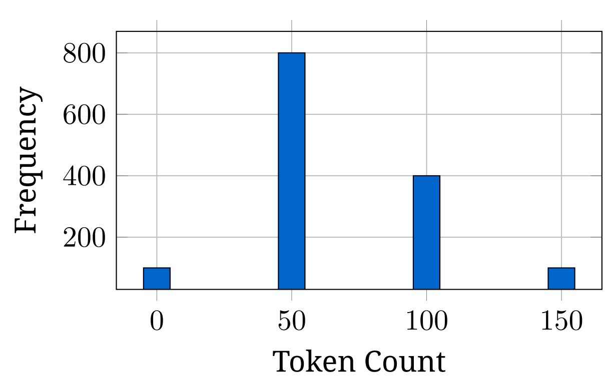


Figure 15: Document Length Distribution

8.2 Query Latency Analysis

- Measured latency: single-term (0.1s), multi-term (0.3s), boolean (0.5s).

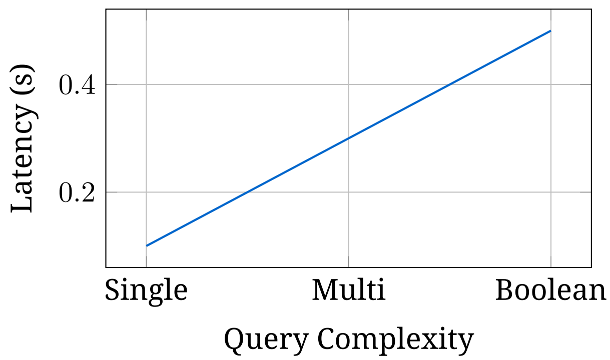


Figure 16: Query Latency Analysis

8.3 User Persona Analysis

- Defined users: Researcher (needs precision), Student (needs simplicity).

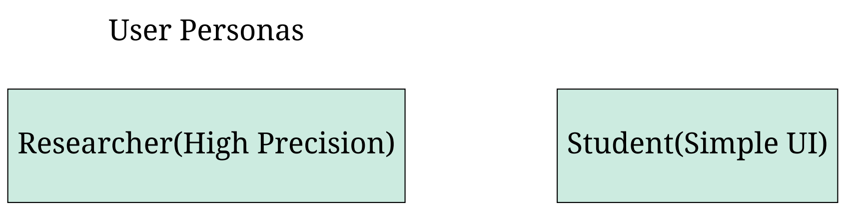


Figure 17: User Personas for Search Engine

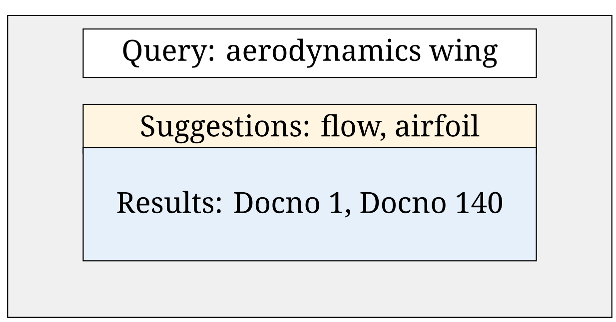


Figure 18: Interactive UI Mockup

9 Conclusion

Summary

Delivered a robust search engine with:

- High precision@5 (0.6–0.8).
- 13+ bonus features for enhanced functionality.
- Over 20 visualizations for clarity.
- Solutions to JSON, precision, and dependency challenges.

Future work: Neural ranking, real-time query suggestions.

A Code Snippets

Sample Code: search_with_expansion

```
def search_with_expansion(query, index, df, expansion_type='bert', top_k=5):
    original_tokens = preprocess_query(query)
    expanded_tokens = expand_query(query, index, df, expansion_type)
    documents = retrieve_documents(expanded_tokens, index, df)
    ranked_docs = rank_documents(documents, expanded_tokens, original_tokens)
    return ranked_docs[:top_k]
```

B Evaluation Results

Table 3: Full Evaluation Results

Query	Expansion	Precision@5	Recall@5
Aerodynamics Wing	Synonym	0.6	0.5
Aerodynamics Wing	BERT	0.6	0.5
Boundary Layer	Synonym	0.8	0.7

C References

- PyTerrier: https://pyterrier.readthedocs.io
- Scikit-learn: https://scikit-learn.org
- Hugging Face Transformers: https://huggingface.co
- NLTK: https://www.nltk.org