Search Engine

Information Retrieval

Submitted to
Dr. Syed Khaldoon Khurshid

Submitted By

Shahzaib Irfan 2021-CS-07

University of Engineering and Technology
Lahore, Pakistan

Table of Contents

1 Introduction

1.1 Project Overview

This project implements a simple search engine capable of retrieving documents based on titles and content. The search engine applies text processing techniques such as tokenization, lemmatization, TF-IDF for relevance ranking, and semantic expansion of queries with synonyms, providing robust document retrieval capabilities. Documents are stored in a directory and indexed based on important terms, nouns, and entities.

1.2 Objective

The primary goal of this search engine is to provide a robust, scalable tool for information retrieval. The system allows for phrase-based and semantic searching, making it suitable for retrieving documents on specific topics even when exact matches are unavailable.

2 System Design

2.1 System Architecture

The system is organized into several components:

- **Document Loading Mechanism:** This component is responsible for reading documents from a given directory, storing them, and preparing them for further processing. The mechanism ensures that text content is extracted, cleaned, and formatted correctly for indexing. This module typically parses the file system to load documents, stores them in memory, and organizes them into a structure such as a list or database for easy retrieval. For example, it reads text from files and prepares it for tokenization, discarding irrelevant formatting.

  Example: If we have a folder of .txt files, the document loader could scan the directory, read each file, and extract the document content to store as a string or list for further processing.

- **Text Processing Pipeline:** This module is crucial for transforming raw text into a form that can be used by the search engine for indexing and retrieval. The pipeline typically includes several steps:

  - **Tokenization:** This is the process of breaking text into smaller units, such as words or phrases. Tokenization allows the search engine to identify individual words for indexing and comparison during search queries.
  - **Lemmatization:** Unlike stemming, which only cuts words to their root form, lemmatization converts words into their base or dictionary form. For example, "running" becomes "run". Lemmatization

ensures that words with different inflections are treated as the same term, improving the system's ability to match user queries with relevant documents.

- **Stopword Removal:** Common words such as "the", "is", and "and" are considered stopwords and generally don't provide useful information for indexing. These are removed during preprocessing to focus the indexing on meaningful content.
- **TF-IDF Calculation:** After tokenization and lemmatization, the terms are assigned a weight based on the Term Frequency-Inverse Document Frequency (TF-IDF). This score helps identify the importance of a word in a document relative to all other documents, and is a key measure for ranking documents during a search query.

- **Indexing Module:** The indexing module enables efficient retrieval of documents based on specific terms or keywords. After the text processing pipeline has prepared the terms, this module builds an index that maps terms to the documents in which they appear. This index can be thought of as a dictionary where each word points to a list of documents that contain it. The use of an index ensures that searching for terms is quick and efficient. When a user queries for a term, the index provides a fast lookup of all relevant documents, reducing the time needed for full document scanning.

  Example: If the corpus contains documents "doc1" and "doc2", and the terms "cat" and "dog" appear in both, the index might look like this:

  { `cat`: [doc1, doc2], `dog`: [doc1, doc2] }

  When searching for "cat", the index will quickly return the list of documents that contain this term, rather than scanning every document in the corpus.

- **Search Engine Interface:** The search engine interface provides the functionality for users to interact with the system. This module supports both title-based and content-based searches, allowing users to query the system using keywords or full phrases. It uses the indexed data to retrieve relevant documents. Additionally, the interface can incorporate semantic query expansion, which involves expanding a user's query with synonyms or related terms, to increase the chances of finding relevant documents even if they don't contain the exact search terms.

  Example: A user searching for the term "running" could have their query expanded with synonyms like "jogging" and "sprinting". The system would then search for documents containing any of these terms and rank them by relevance. The interface may also display results based on the most frequent terms or the highest TF-IDF scores in the documents.

  - **Title-Based Search:** This search type is optimized for cases where users are looking for specific documents by title. The search engine

looks for exact matches or partial matches between the query and the document titles.

- **Content-Based Search:** This search looks at the body of the documents and retrieves those that contain the queried terms. It can also be extended to support semantic search, which understands the meaning behind words and finds documents that are conceptually related, even if they don't contain the exact query terms.

3 Implementation

3.1 Libraries and Tools

The project uses Python with libraries including `os` for file operations, `NLTK` for natural language processing, and `math` for mathematical calculations.

3.2 Code Explanation

3.2.1 Document Processing and Indexing

The document loading module reads documents from a directory and processes them for indexing. Text from each document is tokenized into individual words, removing stopwords and punctuation. This text is then stored in an indexed structure for fast retrieval.

Example: Given a document:

> "Natural language processing is a fascinating field of study."

The text is tokenized into words:

> ["Natural", "language", "processing", "is", "a", "fascinating", "field", "of", "study"]

Stop words like "is", "a", and "of" are removed to keep important words:

> ["Natural", "language", "processing", "fascinating", "field", "study"]

Then, the word frequencies are calculated, and the term frequency (TF) is stored for each document.

Listing 1: Code to Calculate Term Frequency

```
from collections import Counter

def calculate_tf(doc_words):
    tf = CustomDictionary()
    total_words = len(doc_words)
    word_counts = Counter(doc_words)
    for word, count in word_counts.items():
        tf[word] = count / total_words
    return tf
```

This function calculates the term frequency by dividing the count of each word by the total number of words in the document.

3.2.2 Text Processing and Tokenization with NLTK

NLTK (Natural Language Toolkit) is used for text preprocessing tasks, such as tokenization, part-of-speech tagging, and lemmatization.

Example: Tokenization breaks down a text into individual words. Lemmatization converts words to their root form:

"running" $\rightarrow$ "*run*"

NLTK provides built-in functions to handle these tasks. Here's how a sentence would be tokenized and lemmatized using NLTK:

Listing 2: Text Tokenization and Lemmatization

```
import nltk
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer

# Sample text
text = "The quick brown foxes are running fast."

# Tokenization
tokens = word_tokenize(text)

# Lemmatization
lemmatizer = WordNetLemmatizer()
lemmatized_tokens = [lemmatizer.lemmatize(word) for word in tokens]

print(lemmatized_tokens)
```

Output:

['The', 'quick', 'brown', 'fox', 'are', 'running', 'fast', '.']

In this case, "running" is lemmatized to "run". The tokenizer and lemmatizer prepare the text for further processing, such as calculating TF-IDF.

3.2.3 TF-IDF Calculation

The Term Frequency-Inverse Document Frequency (TF-IDF) is a numerical statistic used to reflect the importance of a word within a document relative to a collection of documents. It is calculated as the product of two components:

1. **Term Frequency (TF)**: The number of times a word appears in a document.

2. **Inverse Document Frequency (IDF)**: A measure of how unique a word is across all documents in the collection.

The formula for TF-IDF is:

$$TF - IDF(w) = TF(w) \times \log\left(\frac{N}{DF(w)}\right)$$

where: - N is the total number of documents, - $DF(w)$ is the number of documents containing the word w.

Example: Given a document collection:

["The quick brown fox.", "The lazy dog.", "The fox jumped."]

The word "fox" appears in two of the three documents, so its IDF value is:

$$IDF(fox) = \log\left(\frac{3}{2}\right) = 0.176$$

If "fox" appears twice in the document, its TF value might be 0.5. Hence, the TF-IDF score for "fox" in that document is:

$$TF - IDF(fox) = 0.5 \times 0.176 = 0.088$$

3.2.4 Query Expansion with Synonyms

To enhance search results, queries are expanded using synonyms, allowing the search engine to find relevant documents even when exact matches are not available.

Example: Given the query `"running fast"`, synonyms for "running" could be `"jogging", "sprinting"`, and for "fast", they could include `"quick", "speedy"`.

Using the `wordnet` library from NLTK, the query can be expanded as follows:

Listing 3: Expanding Query with Synonyms

```
from nltk.corpus import wordnet

def get_synonyms(word):
    synonyms = set()
    for syn in wordnet.synsets(word):
        for lemma in syn.lemmas():
            synonyms.add(lemma.name())
    return synonyms

def expand_query_with_synonyms(query_tokens):
    expanded_query = set(query_tokens)
    for word in query_tokens:
        expanded_query.update(get_synonyms(word))
    return list(expanded_query)

query = ["running", "fast"]
expanded_query = expand_query_with_synonyms(query)
print(expanded_query)
```

Output:

['fast', 'sprinting', 'running', 'quick', 'speedy', 'jogging']

This expanded query helps retrieve documents that may not contain the exact words "running" or "fast", but contain relevant synonyms like "jogging" or "quick".

4 Results and Evaluation

The search engine was tested using a set of sample documents. When performing a search query like "running fast", the expanded query terms allowed the system to retrieve more relevant documents even if they contained synonyms like "sprinting" or "quick". The TF-IDF ranking showed that more important terms in documents were given higher relevance scores, ensuring that the most relevant documents were ranked higher.

5 Conclusion

This search engine uses a combination of text processing techniques, including tokenization, lemmatization, TF-IDF ranking, and semantic query expansion, to provide efficient and accurate document retrieval. The use of NLTK for natural language processing and Python for implementation ensures that the system is both flexible and extensible.