Information Retrieval Project

Student Name: Theepan Kumar Gandhi - A20545060

GitHub Repo https://github.com/Theepankumargandhi/Information-Retrieval-Final_project

---

Project Overview

1. **Web crawler** – Scrapy crawler to collect HTML documents (demo corpus)
2. **Document indexer** – TF-IDF based indexing using scikit-learn
3. **Query processor** – Ranked retrieval using cosine similarity

Technologies used

- **Python 3**
- **Scrapy** – web crawling
- **BeautifulSoup4** – HTML parsing
- **scikit-learn** – TF-IDF vectorization
- **Flask** – REST API framework

Information Retrieval Search Engine Project

- Wikipedia crawl + indexing
- Official 3-document TF-IDF search engine
- Query processing to generate `results.csv`

In [1]:
```
# import libraries
import os
import json
import csv
from pathlib import Path
import re
from bs4 import BeautifulSoup
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.feature_extraction.text import TfidfVectorizer
import numpy as np
from flask import Flask, request, jsonify
from IPython.display import display, HTML
import warnings
warnings.filterwarnings('ignore')
```

In [2]:
```
# Creating directories for the project
directories = [
    'data/demo_corpus',      # wikipedia crawl
    'data/html_corpus',      # Fo 3 HTML files
    'data/output'            # For wiki_index.json, index.json and results.csv
]

for directory in directories:
    Path(directory).mkdir(parents=True, exist_ok=True)
    print(f"Directory ready: {directory}")
```

```
Directory ready: data/demo_corpus
Directory ready: data/html_corpus
Directory ready: data/output
```

Web Crawler

The crawler is implemented using **Scrapy**.

Crawler Specifications:

- **https://en.wikipedia.org/wiki/Information_retrieval**
- **Depth Limit:** 2 levels
- **Page Limit:** 100 pages
- **Output:** HTML files saved to `data/demo_corpus/`

HTML text extraction function

In [3]:
```
def extract_text_from_html(html_file_path):

    """Extract clean text from HTML file."""
    try:
        with open(html_file_path, 'r', encoding='utf-8', errors='ignore') as file:
            html_content = file.read()
        soup = BeautifulSoup(html_content, 'lxml')   # Parsing HTML with BeautifulS


        for script in soup(['script', 'style', 'meta', 'link']):
            script.decompose()

        text = soup.get_text(separator=' ')  # Clean the text

        # Removing extra whitespace
        text = re.sub(r'\s+', ' ', text)
        text = text.strip()

        return text
    except Exception as e:
        print(f"Error{html_file_path}: {e}")
        return ""
```

Wikipedia Crawler – Collects up to 100 pages from the Information Retrieval Wikipedia branch.

In [4]:
```python
# Scrapy crawler for collecting up to 100 Wikipedia pages on Information Retrieval
import scrapy
from pathlib import Path
from scrapy.crawler import CrawlerProcess

class WikipediaIRSpider(scrapy.Spider):
    name = "wikipedia_ir"
    start_urls = ["https://en.wikipedia.org/wiki/Information_retrieval"]

    custom_settings = {
        "DEPTH_LIMIT": 2,
        "CLOSESPIDER_PAGECOUNT": 100,
        "ROBOTSTXT_OBEY": True,
        "DOWNLOAD_DELAY": 1.0,
        "AUTOTHROTTLE_ENABLED": True,
        "AUTOTHROTTLE_START_DELAY": 1.0,
        "AUTOTHROTTLE_MAX_DELAY": 5.0,
        "AUTOTHROTTLE_TARGET_CONCURRENCY": 1.0,
        "LOG_LEVEL": "INFO",
        "USER_AGENT": "IRCourseCrawler/1.0 (student project)"
    }

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.output_dir = Path("data/demo_corpus")
        self.output_dir.mkdir(parents=True, exist_ok=True)
        self.page_counter = 0

    def parse(self, response):
        # Save using unique incremental filenames
        self.page_counter += 1
        page_id = f"page_{self.page_counter:03d}"

        filename = self.output_dir / f"{page_id}.html"
        filename.write_bytes(response.body)
        self.logger.info(f"Saved: {filename} ({response.url})")

        # Follow other Wikipedia links
        for href in response.css("a::attr(href)").getall():
            if href.startswith("/wiki/") and not any(prefix in href for prefix in [
                yield scrapy.Request(response.urljoin(href), callback=self.parse)


def run_wiki_crawler():
    process = CrawlerProcess()
    process.crawl(WikipediaIRSpider)
    process.start()

# run_wiki_crawler()
```

Sample print of wikipedia crawled page

In [5]:
```python
from pathlib import Path

wiki_corpus_path = Path("data/demo_corpus")
wiki_html_files_all = sorted(wiki_corpus_path.glob("*.html"))

print(f"Total HTML files physically in demo_corpus: {len(wiki_html_files_all)}")

# Use only the first 100 for indexing (professor's max constraint)
wiki_html_files = wiki_html_files_all[:100]

print(f"Number of files we will index for Wikipedia corpus: {len(wiki_html_files)}"
print("\nFirst 10 files used for indexing:")
for f in wiki_html_files[:10]:
    print("  ", f.name)

# Preview of the first HTML document
if wiki_html_files:
    first_file = wiki_html_files[0]
    print("\nPreview of first HTML page:")
    print("File:", first_file.name)
    sample_text = extract_text_from_html(first_file)

    print("\nExtracted Text (first 500 characters):")
    print(sample_text[:500], "...")
else:
    print("No HTML files found in the folder.")
```

```
Total HTML files physically in demo_corpus: 100
Number of files we will index for Wikipedia corpus: 100

First 10 files used for indexing:
  Adversarial_information_retrieval.html
  Artificial_intelligence_content_detection.html
  Audio_search_engine.html
  Automatic_summarization.html
  Bibliographic_databases.html
  C._J._van_Rijsbergen.html
  Capacity_planning.html
  Collaborative_search_engine.html
  Compound_term_processing.html
  Computer-assisted_legal_research.html

Preview of first HTML page:
File: Adversarial_information_retrieval.html

Extracted Text (first 500 characters):
Adversarial information retrieval - Wikipedia Jump to content Main menu Main menu mo
ve to sidebar hide Navigation Main page Contents Current events Random article About
Wikipedia Contact us Contribute Help Learn to edit Community portal Recent changes U
pload file Special pages Search Search Appearance Donate Create account Log in Perso
nal tools Donate Create account Log in Contents move to sidebar hide (Top) 1 Topics
2 History 3 See also 4 References 5 External links Toggle the table of contents ...
```

Load and preview cleaned text from the crawled Wikipedia pages.

In [6]:
```python
# Build document dictionary for crawled Wikipedia corpus

wiki_documents = {}

print("\nBuilding document dictionary for Wikipedia corpus")
for html_file in wiki_html_files:
    doc_id = html_file.stem  # e.g., page_001, page_002, ...
    text = extract_text_from_html(html_file)  # uses your existing preprocessing
    wiki_documents[doc_id] = text

print(f"Total Wikipedia documents loaded for indexing: {len(wiki_documents)}")

# Show a few sample docs as artifacts
for i, (doc_id, text) in enumerate(wiki_documents.items()):
    if i >= 3:
        break
    print(f"\nDocument ID: {doc_id}")
    print(f" Text length: {len(text)} characters")
    print(f" Preview: {text[:300]}")
```

```
Building document dictionary for Wikipedia corpus
Total Wikipedia documents loaded for indexing: 100

Document ID: Adversarial_information_retrieval
 Text length: 4103 characters
 Preview: Adversarial information retrieval - Wikipedia Jump to content Main menu Ma
in menu move to sidebar hide Navigation Main page Contents Current events Random art
icle About Wikipedia Contact us Contribute Help Learn to edit Community portal Recen
t changes Upload file Special pages Search Search Appearan

Document ID: Artificial_intelligence_content_detection
 Text length: 18033 characters
 Preview: Artificial intelligence content detection - Wikipedia Jump to content Main
menu Main menu move to sidebar hide Navigation Main page Contents Current events Ran
dom article About Wikipedia Contact us Contribute Help Learn to edit Community porta
l Recent changes Upload file Special pages Search Search

Document ID: Audio_search_engine
 Text length: 9096 characters
 Preview: Audio search engine - Wikipedia Jump to content Main menu Main menu move t
o sidebar hide Navigation Main page Contents Current events Random article About Wik
ipedia Contact us Contribute Help Learn to edit Community portal Recent changes Uplo
ad file Special pages Search Search Appearance Donate Crea
```

Build the TF-IDF index for the 100-page Wikipedia corpus and save it as wiki_index.json.

In [7]:
```python
# TF-IDF index for the Wikipedia corpus
from sklearn.feature_extraction.text import TfidfVectorizer
import numpy as np
import json
import os
from pathlib import Path
```

```python
print("\nBuilding TF-IDF index for Wikipedia corpus")

wiki_doc_ids = list(wiki_documents.keys())
wiki_texts = [wiki_documents[doc_id] for doc_id in wiki_doc_ids]

print(f"Documents: {len(wiki_doc_ids)}")

wiki_vectorizer = TfidfVectorizer(
    lowercase=True,
    stop_words='english',
    max_features=None,
    norm='l2'
)

wiki_tfidf_matrix = wiki_vectorizer.fit_transform(wiki_texts)
wiki_feature_names = wiki_vectorizer.get_feature_names_out()

print("TF-IDF matrix created for Wikipedia corpus")
print(f"Documents: {wiki_tfidf_matrix.shape[0]}")
print(f"Vocabulary size: {wiki_tfidf_matrix.shape[1]} terms")
print(f"Matrix shape: {wiki_tfidf_matrix.shape}")

sparsity = 1 - wiki_tfidf_matrix.nnz / (wiki_tfidf_matrix.shape[0] * wiki_tfidf_mat
print(f"Sparsity: {sparsity * 100:.2f}%")

# Pack index in a JSON-friendly structure
wiki_index_data = {
    "document_ids": wiki_doc_ids,
    "vocabulary": wiki_feature_names.tolist(),
    "tfidf_matrix": wiki_tfidf_matrix.toarray().tolist(),
    "vectorizer_params": {
        "lowercase": True,
        "stop_words": "english",
        "norm": "l2"
    }
}

wiki_index_path = Path("data/output/wiki_index.json")
wiki_index_path.parent.mkdir(parents=True, exist_ok=True)

with open(wiki_index_path, "w", encoding="utf-8") as f:
    json.dump(wiki_index_data, f, indent=2)

print("\nWikipedia index saved")
print(f"Location: {wiki_index_path}")
print(f"File size: {os.path.getsize(wiki_index_path) / 1024:.2f} KB")
print(f"Index contains:")
print(f" - {len(wiki_index_data['document_ids'])} documents")
print(f" - {len(wiki_index_data['vocabulary'])} vocabulary terms")
print(
    f" - TF-IDF matrix: "
    f"{len(wiki_index_data['tfidf_matrix'])} x {len(wiki_index_data['tfidf_matrix']
)
```

```
Building TF-IDF index for Wikipedia corpus
Documents: 100
TF-IDF matrix created for Wikipedia corpus
Documents: 100
Vocabulary size: 25895 terms
Matrix shape: (100, 25895)
Sparsity: 96.05%

Wikipedia index saved
Location: data\output\wiki_index.json
File size: 32455.62 KB
Index contains:
 - 100 documents
 - 25895 vocabulary terms
 - TF-IDF matrix: 100 x 25895
```

Load the saved Wikipedia TF-IDF index and display a few sample entries.

In [8]:
```python
#few sample entries from the Wikipedia index JSON

import json
from pathlib import Path
wiki_index_path = Path("data/output/wiki_index.json")
print("Loading Wikipedia TF-IDF index from:", wiki_index_path)
with open(wiki_index_path, "r", encoding="utf-8") as f:
    wiki_index = json.load(f)
print("\nIndex Summary")
print("Documents:", len(wiki_index["document_ids"]))
print("Vocabulary size:", len(wiki_index["vocabulary"]))
print("TF-IDF matrix shape:", len(wiki_index["tfidf_matrix"]), "x", len(wiki_index[

#couple of document IDs as a sample
print("\nSample document IDs:")
for doc_id in wiki_index["document_ids"][:5]:
    print(" -", doc_id)

#few vocabulary terms
print("\nSample vocabulary terms:")
for term in wiki_index["vocabulary"][:15]:
    print(" -", term)
print("\nTF-IDF vector sample for the first document:")
first_vec = wiki_index["tfidf_matrix"][0]
print("Length:", len(first_vec))
print("First 20 values:", first_vec[:20])
```

```
Loading Wikipedia TF-IDF index from: data\output\wiki_index.json

Index Summary
Documents: 100
Vocabulary size: 25895
TF-IDF matrix shape: 100 x 25895

Sample document IDs:
 - Adversarial_information_retrieval
 - Artificial_intelligence_content_detection
 - Audio_search_engine
 - Automatic_summarization
 - Bibliographic_databases

Sample vocabulary terms:
 - 00
 - 000
 - 000000000103
 - 00001
 - 00009
 - 0001
 - 00011
 - 00013
 - 00018
 - 00019
 - 0002
 - 00020
 - 0003
 - 00037
 - 00051

TF-IDF vector sample for the first document:
Length: 25895
First 20 values: [0.03421126546866168, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

Document Indexer

The indexer processes HTML documents and builds a TF-IDF based vector space model for information retrieval.

Input:

- **3 HTML files** from `data/html_corpus/`:
  - `0F64A61C-DF01-4F43-8B8D-F0319C41768E.html`
  - `1F648A7F-2C64-458C-BFAF-463A071530ED.html`
  - `6B3BD97C-DEF2-49BB-B2B6-80F2CD53C4D3.html`

Process:

HTML Pages → HTML Parsing → Clean Text → TF-IDF Vectorization → Document–Term Matrix → index.json

Output:

- index.json - Contains TF-IDF data and document information for query processing
- HTML Parser - Extracts text from HTML documents
- TF-IDF Vectorizer - Converts documents to weighted term vectors
- Index Builder - Serializes the model to JSON format

HTML Parsing and Clean Text Extraction

In [9]:
```
official_files = [
    '0F64A61C-DF01-4F43-8B8D-F0319C41768E.html',
    '1F648A7F-2C64-458C-BFAF-463A071530ED.html',
    '6B3BD97C-DEF2-49BB-B2B6-80F2CD53C4D3.html'
]
corpus_path = Path('data/html_corpus')
print("Checking files")
documents = {}
for filename in official_files:
    file_path = corpus_path / filename
    if file_path.exists():
        doc_id = filename.replace('.html', '')   # Extract document ID from filenam
        text = extract_text_from_html(file_path)  # Extract text content
        documents[doc_id] = text
        print(f"Loaded: {filename}")
        print(f"  Text length: {len(text)} characters")
    else:
        print(f"Missing: {filename}")
print(f"\nTotal documents loaded: {len(documents)}")
```

```
Checking files
Loaded: 0F64A61C-DF01-4F43-8B8D-F0319C41768E.html
  Text length: 58606 characters
Loaded: 1F648A7F-2C64-458C-BFAF-463A071530ED.html
  Text length: 80858 characters
Loaded: 6B3BD97C-DEF2-49BB-B2B6-80F2CD53C4D3.html
  Text length: 38748 characters

Total documents loaded: 3
```

Build TF-IDF vectors for the three given documents.

In [10]:
```
# Build TF-IDF vectors for  documents
print("Building TF-IDF Index")
doc_ids = list(documents.keys())
doc_texts = [documents[doc_id] for doc_id in doc_ids]
```

```
vectorizer = TfidfVectorizer(                # Create TF-IDF vectorizer
    lowercase=True,
    stop_words='english',
    max_features=None,
    norm='l2'  # L2 normalization for cosine similarity
)
tfidf_matrix = vectorizer.fit_transform(doc_texts)  # Fit_transform documents
feature_names = vectorizer.get_feature_names_out()  # Get feature names (terms)
print(f"TF-IDF matrix created")
print(f"Documents: {tfidf_matrix.shape[0]}")
print(f"Vocabulary size: {tfidf_matrix.shape[1]} unique terms")
print(f"Matrix shape: {tfidf_matrix.shape}")
print(f"Matrix sparsity: {(1 - tfidf_matrix.nnz / (tfidf_matrix.shape[0] * tfidf_ma
```

```
Building TF-IDF Index
TF-IDF matrix created
Documents: 3
Vocabulary size: 4692 unique terms
Matrix shape: (3, 4692)
Matrix sparsity: 53.71%
```

Building the Inverted Index (index.json)

In [11]:
```
# Create the index structure
index_data = {
    'document_ids': doc_ids,
    'vocabulary': feature_names.tolist(),
    'tfidf_matrix': tfidf_matrix.toarray().tolist(),  # Convert sparse matrix to li
    'vectorizer_params': {
        'lowercase': True,
        'stop_words': 'english',
        'norm': 'l2'
    }
}

index_file_path = 'data/output/index.json'     # Save index to JSON file
with open(index_file_path, 'w', encoding='utf-8') as f:
    json.dump(index_data, f, indent=2)
print("Index saved successfully")
print(f"Location: {index_file_path}")
print(f"File size: {os.path.getsize(index_file_path) / 1024:.2f} KB")
print(f"\nIndex contains:")
print(f"  - {len(index_data['document_ids'])} documents")
print(f"  - {len(index_data['vocabulary'])} vocabulary terms")
print(f"  - TF-IDF matrix: {len(index_data['tfidf_matrix'])} x {len(index_data['tfi
```

```
Index saved successfully
Location: data/output/index.json
File size: 348.69 KB

Index contains:
  - 3 documents
  - 4692 vocabulary terms
  - TF-IDF matrix: 3 x 4692
```

Part 3: Query Processor

The query processor loads the index and performs ranked retrieval using cosine similarity.

Input:

- index.json - The TF-IDF index created above
- queries.csv - Query file with columns: `query_id`, `query_text`

Process:

Queries → TF-IDF Vectorization → Cosine Similarity Computation → Document Ranking → Top-K Results

Output:

- results.csv - Ranked results with columns: `query_id`, `rank`, `document_id`

Ranking Method:

- **Cosine Similarity** - Measures the angle between query and document vectors
- Documents are ranked in descending order of similarity
- All 3 documents are ranked for each query

In this step, I load the saved TF-IDF index (index.json) and reconstruct the document IDs, vocabulary, and matrix for searching. I also load the instructor-provided queries.csv file and print a quick preview to confirm everything is ready for ranking.

In [12]:
```python
# Load the index from JSON
print("Loading index and queries")
with open('data/output/index.json', 'r', encoding='utf-8') as f:  # Load index.json
    loaded_index = json.load(f)
# Reconstruct the TF-IDF components
loaded_doc_ids = loaded_index['document_ids']
loaded_vocabulary = loaded_index['vocabulary']
loaded_tfidf_matrix = np.array(loaded_index['tfidf_matrix'])

print(f"Index loaded successfully")
print(f"  Documents: {len(loaded_doc_ids)}")
print(f"  Vocabulary: {len(loaded_vocabulary)} terms")
queries = []                                        # Load queries from CSV
with open('queries.csv', 'r', encoding='utf-8') as f:
    reader = csv.DictReader(f)
    for row in reader:
        queries.append({
            'query_id': row['query_id'],
            'query_text': row['query_text']
        })
```

```
print(f"\nQueries loaded successfully")
print(f"Total queries: {len(queries)}")
print("\nQuery preview:")
for q in queries:
    print(f"  - {q['query_id']}: '{q['query_text']}'")
```

```
Loading index and queries
Index loaded successfully
  Documents: 3
  Vocabulary: 4692 terms

Queries loaded successfully
Total queries: 3

Query preview:
  - 6E93CDD1-52F9-4F41-A405-54E398EF6FF8: 'information overload'
  - 0D97BCC6-C46E-4242-9777-7CEAED55B362: 'database server hardware specs'
  - 78452FF4-94D7-422C-9283-A14615C44ADC: 'search engine open sorce'
```

In [13]:
```
def process_query(query_text, vocabulary, tfidf_matrix, doc_ids):
    """Process query and return ranked documents by similarity score."""
    # Create a vectorizer with the same vocabulary
    query_vectorizer = TfidfVectorizer(
        lowercase=True,
        stop_words='english',
        vocabulary=vocabulary,
        norm='l2'
    )
    query_vector = query_vectorizer.fit_transform([query_text]).toarray()  # Transf
    similarities = cosine_similarity(query_vector, tfidf_matrix)[0]  # Compute cosi
    doc_scores = list(zip(doc_ids, similarities))
    doc_scores.sort(key=lambda x: x[1], reverse=True)   # Sort by score in descendi
    # Add rank (1-indexed)
    ranked_results = [(doc_id, rank + 1, score)
                      for rank, (doc_id, score) in enumerate(doc_scores)]

    return ranked_results
print("Query processing function")
```

```
Query processing function
```

For each query, I convert the text into a TF-IDF vector using the same vocabulary as the index. I compute cosine similarity with all documents, rank them, print the top results, and save everything into results.csv.

In [14]:
```
# Process all queries and collect results
print("Processing queries")
all_results = []

for query in queries:
    query_id = query['query_id']
    query_text = query['query_text']

    print(f"\nQuery: '{query_text}'")
    ranked_docs = process_query(           # Ranked results
        query_text,
        loaded_vocabulary,
        loaded_tfidf_matrix,
        loaded_doc_ids
    )

    print(f"  Results:")            # Display results
    for doc_id, rank, score in ranked_docs:
        print(f"    Rank {rank}: {doc_id} (score: {score:.4f})")
        all_results.append({
            'query_id': query_id,
            'rank': rank,
            'document_id': doc_id
        })
# Save results to CSV
results_file_path = 'data/output/results.csv'
with open(results_file_path, 'w', newline='', encoding='utf-8') as f:
    writer = csv.DictWriter(f, fieldnames=['query_id', 'rank', 'document_id'])
    writer.writeheader()
    writer.writerows(all_results)

print(f"Results saved to: {results_file_path}")
print(f"Total result entries: {len(all_results)}")
```

```
Processing queries

Query: 'information overload'
  Results:
    Rank 1: 6B3BD97C-DEF2-49BB-B2B6-80F2CD53C4D3 (score: 0.3643)
    Rank 2: 0F64A61C-DF01-4F43-8B8D-F0319C41768E (score: 0.0735)
    Rank 3: 1F648A7F-2C64-458C-BFAF-463A071530ED (score: 0.0688)

Query: 'database server hardware specs'
  Results:
    Rank 1: 1F648A7F-2C64-458C-BFAF-463A071530ED (score: 0.3688)
    Rank 2: 0F64A61C-DF01-4F43-8B8D-F0319C41768E (score: 0.0225)
    Rank 3: 6B3BD97C-DEF2-49BB-B2B6-80F2CD53C4D3 (score: 0.0157)

Query: 'search engine open sorce'
  Results:
    Rank 1: 0F64A61C-DF01-4F43-8B8D-F0319C41768E (score: 0.5579)
    Rank 2: 6B3BD97C-DEF2-49BB-B2B6-80F2CD53C4D3 (score: 0.1255)
    Rank 3: 1F648A7F-2C64-458C-BFAF-463A071530ED (score: 0.0308)
Results saved to: data/output/results.csv
Total result entries: 9
```

Display the generated results.csv with ranked outputs.

```
In [15]: # Display the generated results.csv
print("Generated Results")
with open('data/output/results.csv', 'r', encoding='utf-8') as f:
    reader = csv.DictReader(f)
    results_data = list(reader)

# Display first 10 rows
#print("\nFirst 10 rows of results.csv:")
print(f"{'Query ID':<40} {'Rank':<6} {'Document ID':<40}")
```

```
for row in results_data[:10]:
    print(f"{row['query_id']:<40} {row['rank']:<6} {row['document_id']:<40}")

print(f"\nshowing total of {len(results_data)} rows")
```

```
Generated Results
Query ID                                 Rank   Document ID
6E93CDD1-52F9-4F41-A405-54E398EF6FF8     1      6B3BD97C-DEF2-49BB-B2B6-80F2CD53C4D3
6E93CDD1-52F9-4F41-A405-54E398EF6FF8     2      0F64A61C-DF01-4F43-8B8D-F0319C41768E
6E93CDD1-52F9-4F41-A405-54E398EF6FF8     3      1F648A7F-2C64-458C-BFAF-463A071530ED
0D97BCC6-C46E-4242-9777-7CEAED55B362     1      1F648A7F-2C64-458C-BFAF-463A071530ED
0D97BCC6-C46E-4242-9777-7CEAED55B362     2      0F64A61C-DF01-4F43-8B8D-F0319C41768E
0D97BCC6-C46E-4242-9777-7CEAED55B362     3      6B3BD97C-DEF2-49BB-B2B6-80F2CD53C4D3
78452FF4-94D7-422C-9283-A14615C44ADC     1      0F64A61C-DF01-4F43-8B8D-F0319C41768E
78452FF4-94D7-422C-9283-A14615C44ADC     2      6B3BD97C-DEF2-49BB-B2B6-80F2CD53C4D3
78452FF4-94D7-422C-9283-A14615C44ADC     3      1F648A7F-2C64-458C-BFAF-463A071530ED

showing total of 9 rows
```

Sample query results and their relevance

In [16]:

```
# Analyze a sample query in detail
print("Detailed Query Analysis")
# Pick the first query for detailed analysis
sample_query = queries[0]
query_id = sample_query['query_id']
query_text = sample_query['query_text']

print(f"\nQuery ID: {query_id}")
print(f"Query Text: '{query_text}'")
print("\nQuery Terms: ", query_text.lower().split())

# Process the query
ranked_docs = process_query(
    query_text,
    loaded_vocabulary,
    loaded_tfidf_matrix,
    loaded_doc_ids
)

print(f"\n{'Rank':<6} {'Document ID':<40} {'Score':<10}")
for doc_id, rank, score in ranked_docs:
    print(f"{rank:<6} {doc_id:<40} {score:.6f}")
# Show which query terms are in the vocabulary
query_terms = [term for term in query_text.lower().split() if term in loaded_vocabu
print(f"\nQuery terms found in vocabulary: {query_terms}")
```

```
Detailed Query Analysis

Query ID: 6E93CDD1-52F9-4F41-A405-54E398EF6FF8
Query Text: 'information overload'

Query Terms:  ['information', 'overload']

Rank   Document ID                              Score
1      6B3BD97C-DEF2-49BB-B2B6-80F2CD53C4D3     0.364314
2      0F64A61C-DF01-4F43-8B8D-F0319C41768E     0.073532
3      1F648A7F-2C64-458C-BFAF-463A071530ED     0.068816

Query terms found in vocabulary: ['information', 'overload']
```

Display detailed statistics of the indexed documents, vocabulary, TF-IDF matrix, and query results.

In [17]:
```python
# Display comprehensive index statistics
print("Index Statistics")
# Document statistics
print(f"\nDocument Statistics:")
print(f"  Total documents indexed: {len(loaded_doc_ids)}")
print(f"  Document IDs:")
for i, doc_id in enumerate(loaded_doc_ids, 1):
    print(f"    {i}. {doc_id}")

# Vocabulary statistics
print(f"\nVocabulary Statistics:")
print(f"  Total unique terms: {len(loaded_vocabulary)}")
print(f"  Sample terms (first 20): {loaded_vocabulary[:20]}")

# TF-IDF matrix statistics
print(f"\nTF-IDF Matrix:")
print(f"  Shape: {loaded_tfidf_matrix.shape[0]} documents × {loaded_tfidf_matrix.sh
print(f"  Non-zero entries: {np.count_nonzero(loaded_tfidf_matrix)}")
print(f"  Sparsity: {(1 - np.count_nonzero(loaded_tfidf_matrix) / loaded_tfidf_matr

# Query statistics
print(f"\nQuery Statistics:")
print(f"  Total queries processed: {len(queries)}")
print(f"  Results per query: {len(loaded_doc_ids)} (all documents ranked)")
```

```
Index Statistics

Document Statistics:
  Total documents indexed: 3
  Document IDs:
    1. 0F64A61C-DF01-4F43-8B8D-F0319C41768E
    2. 1F648A7F-2C64-458C-BFAF-463A071530ED
    3. 6B3BD97C-DEF2-49BB-B2B6-80F2CD53C4D3

Vocabulary Statistics:
  Total unique terms: 4692
  Sample terms (first 20): ['00063', '00065', '0020', '004', '0086', '01', '012', '0
131580183', '0131964280', '0138613372', '016555159802400509', '0167', '0197224005013
3634', '02', '020', '0201741308', '025', '0252087127', '0257', '0271']

TF-IDF Matrix:
  Shape: 3 documents × 4692 terms
  Non-zero entries: 6516
  Sparsity: 53.71%

Query Statistics:
  Total queries processed: 3
  Results per query: 3 (all documents ranked)
```

Optional Enhancements

I implement two optional features:

1. **Spelling Correction (NLTK)**
    - Detects and corrects misspelled query terms, It uses Uses dictionary-based approach

In [18]:
```python
# Import NLTK and download required resources
import nltk
from nltk.corpus import wordnet
from nltk.metrics.distance import edit_distance
nltk.download('wordnet', quiet=True)
nltk.download('omw-1.4', quiet=True)
nltk.download('words', quiet=True)    # English word list

from nltk.corpus import words
english_words = set(words.words())

print("NLTK resources downloaded successfully")
print(f"English vocabulary loaded: {len(english_words)} words")
```

```
NLTK resources downloaded successfully
English vocabulary loaded: 235892 words
```

Spelling correction module using NLTK to fix misspelled query terms.

In [19]:
```python
def correct_spelling(word):
    """Return corrected spelling or original word if correct."""
    word_lower = word.lower()
    if word_lower in english_words:   # If word is already correct, return it
        return word_lower

    candidates = [w for w in english_words    # Find the closest match using edit d
                  if abs(len(w) - len(word_lower)) <= 2]

    if not candidates:
        return word_lower

    closest_word = min(candidates,  # Find word with minimum edit distance
                       key=lambda w: edit_distance(word_lower, w))

    if edit_distance(word_lower, closest_word) <= 2:
        return closest_word

    return word_lower

# Test spelling correction
print("Spelling Correction Examples:")
test_words = ['informaton', 'search', 'engine', 'retrieval']
for word in test_words:
    corrected = correct_spelling(word)
    status = "Corrected" if word != corrected else "Already correct"
    print(f"  {word:15} → {corrected:15} ({status})")
```

```
Spelling Correction Examples:
  informaton      → information     (Corrected)
  search          → search          (Already correct)
  engine          → engine          (Already correct)
  retrieval       → retrieval       (Already correct)
```

Semantic Search using Word2Vec Embeddings

Word2Vec:

- Uses pre-trained word embeddings to capture semantic relationships
- Documents and queries are represented as dense vectors
- Cosine similarity in embedding space finds semantically related documents
- Model: **glove-wiki-gigaword-50** (50-dimensional vectors, lightweight)
- Cosine similarity between query and document embeddings

In [20]:
```
# Import gensim for word embeddings
import gensim.downloader as api

print("Loading pre-trained word embeddings...")
print("Model: glove-wiki-gigaword-50 (50-dimensional vectors)")
# Load the pretrained model
word2vec_model = api.load('glove-wiki-gigaword-50')

print("Word2Vec model loaded successfully!")
print(f"Vocabulary size: {len(word2vec_model)} words")
print(f"Vector dimensions: {word2vec_model.vector_size}")
# Test with a simple example
```

```python
test_word = "search"
if test_word in word2vec_model:
    similar_words = word2vec_model.most_similar(test_word, topn=5)
    print(f"\nExample - Words similar to '{test_word}':")
    for word, score in similar_words:
        print(f"  {word}: {score:.4f}")
```

```
Loading pre-trained word embeddings...
Model: glove-wiki-gigaword-50 (50-dimensional vectors)
Word2Vec model loaded successfully!
Vocabulary size: 400000 words
Vector dimensions: 50

Example - Words similar to 'search':
  searching: 0.8898
  searches: 0.8053
  information: 0.7864
  finding: 0.7863
  tracking: 0.7722
```

Generate document-level embeddings by averaging Word2Vec vectors for each document.

In [21]:
```python
def get_document_embedding(text, model):
    """Convert document text to embedding by averaging word vectors."""
    words = text.lower().split()

    # Filter words that exist in the model's vocabulary
    word_vectors = []
    for word in words:
        if word in model:
            word_vectors.append(model[word])

    # Return average of word vectors (or zero vector if no words found)
    if len(word_vectors) > 0:
        return np.mean(word_vectors, axis=0)
    else:
        return np.zeros(model.vector_size)

# Create embeddings for all documents
print("Creating document embeddings using Word2Vec...")
doc_embeddings = {}
for doc_id, text in documents.items():
    embedding = get_document_embedding(text, word2vec_model)
    doc_embeddings[doc_id] = embedding

    # Count how many words were found in vocabulary
    words_in_vocab = sum(1 for word in text.lower().split() if word in word2vec_mod
    total_words = len(text.split())
    print(f"{doc_id}")
    print(f"  Words in vocabulary: {words_in_vocab}/{total_words} ({words_in_vocab/


print(f" Document embeddings created: {len(doc_embeddings)} documents")
```

```
Creating document embeddings using Word2Vec...
0F64A61C-DF01-4F43-8B8D-F0319C41768E
  Words in vocabulary: 8124/9643 (84.2%)
1F648A7F-2C64-458C-BFAF-463A071530ED
  Words in vocabulary: 10594/12213 (86.7%)
6B3BD97C-DEF2-49BB-B2B6-80F2CD53C4D3
  Words in vocabulary: 4754/5726 (83.0%)
 Document embeddings created: 3 documents
```

In [22]:
```
def process_query_word2vec(query_text, doc_embeddings, model):
    """Rank documents using Word2Vec semantic similarity."""
    # Get query embedding
    query_embedding = get_document_embedding(query_text, model)

    # Compute cosine similarity with each document
    similarities = []
    for doc_id, doc_embedding in doc_embeddings.items():
        # Compute cosine similarity
        similarity = cosine_similarity(
            query_embedding.reshape(1, -1),
            doc_embedding.reshape(1, -1)
        )[0][0]
        similarities.append((doc_id, similarity))

    # Sort by similarity (descending)
    similarities.sort(key=lambda x: x[1], reverse=True)

    # Add ranks
    ranked_results = [(doc_id, rank + 1, score)
                      for rank, (doc_id, score) in enumerate(similarities)]

    return ranked_results
print("Word2Vec query processing function")
```

```
Word2Vec query processing function
```

Compare TF-IDF and Word2Vec results for all queries

In [23]:
```
# Compare TF-IDF and Word2Vec results for all queries
print("Comparison: TF-IDF vs Word2Vec Semantic Search")
for query in queries:
    query_id = query['query_id']
    query_text = query['query_text']

    print(f"\nQuery: '{query_text}'")
    # Get TF-IDF results
    tfidf_results = process_query(
        query_text,
        loaded_vocabulary,
        loaded_tfidf_matrix,
        loaded_doc_ids
    )
    # Get Word2Vec results
    w2v_results = process_query_word2vec(
        query_text,
        doc_embeddings,
        word2vec_model
    )

    # Display side-by-side comparison
    print(f"{'TF-IDF Ranking':<35} | {'Word2Vec Ranking':<35}")
    for i in range(len(tfidf_results)):
        tfidf_doc, tfidf_rank, tfidf_score = tfidf_results[i]
        w2v_doc, w2v_rank, w2v_score = w2v_results[i]

        tfidf_str = f"Rank {tfidf_rank}: {tfidf_doc[:20]}... ({tfidf_score:.4f})"
        w2v_str = f"Rank {w2v_rank}: {w2v_doc[:20]}... ({w2v_score:.4f})"

        print(f"{tfidf_str:<35} | {w2v_str:<35}")
```

```
Comparison: TF-IDF vs Word2Vec Semantic Search

Query: 'information overload'
TF-IDF Ranking                      | Word2Vec Ranking
Rank 1: 6B3BD97C-DEF2-49BB-B... (0.3643) | Rank 1: 6B3BD97C-DEF2-49BB-B... (0.6950)
Rank 2: 0F64A61C-DF01-4F43-8... (0.0735) | Rank 2: 1F648A7F-2C64-458C-B... (0.6930)
Rank 3: 1F648A7F-2C64-458C-B... (0.0688) | Rank 3: 0F64A61C-DF01-4F43-8... (0.6510)

Query: 'database server hardware specs'
TF-IDF Ranking                      | Word2Vec Ranking
Rank 1: 1F648A7F-2C64-458C-B... (0.3688) | Rank 1: 1F648A7F-2C64-458C-B... (0.5860)
Rank 2: 0F64A61C-DF01-4F43-8... (0.0225) | Rank 2: 6B3BD97C-DEF2-49BB-B... (0.5558)
Rank 3: 6B3BD97C-DEF2-49BB-B... (0.0157) | Rank 3: 0F64A61C-DF01-4F43-8... (0.5157)

Query: 'search engine open sorce'
TF-IDF Ranking                      | Word2Vec Ranking
Rank 1: 0F64A61C-DF01-4F43-8... (0.5579) | Rank 1: 0F64A61C-DF01-4F43-8... (0.7345)
Rank 2: 6B3BD97C-DEF2-49BB-B... (0.1255) | Rank 2: 1F648A7F-2C64-458C-B... (0.7324)
Rank 3: 1F648A7F-2C64-458C-B... (0.0308) | Rank 3: 6B3BD97C-DEF2-49BB-B... (0.7154)
```

Part 4: Flask REST API

REST API for programmatic document search.

Usage: `POST /search` with `{"query": "text", "top_k": 3}`

Returns ranked documents with scores.

In [26]:
```
# Flask API implementation
import threading
import time

app = Flask(__name__)
api_vocabulary = None
api_tfidf_matrix = None
api_doc_ids = None
```

```python
@app.route('/', methods=['GET'])
def home():
    """Home page endpoint."""
    return jsonify({
        'message': 'Flask API is running!',
        'endpoints': {
            '/search': 'POST - Search documents',
            '/health': 'GET - Health check'
        }
    })

@app.route('/search', methods=['POST'])
def search():
    """
    Search endpoint that accepts a query and returns ranked results.
    """
    try:
        # Get request data
        data = request.get_json()
        if not data or 'query' not in data:
            return jsonify({'error': 'Missing query parameter'}), 400
        query_text = data['query']
        top_k = data.get('top_k', 3)  # Default to top 3 results

        ranked_docs = process_query(   # Process the query
            query_text,
            api_vocabulary,
            api_tfidf_matrix,
            api_doc_ids
        )

        results = []        # Format results
        for doc_id, rank, score in ranked_docs[:top_k]:
            results.append({
                'rank': rank,
                'document_id': doc_id,
                'score': float(score)
            })

        return jsonify({
            'query': query_text,
            'results': results
        })

    except Exception as e:
        return jsonify({'error': str(e)}), 500

@app.route('/health', methods=['GET'])
def health():
    """Health check endpoint."""
    return jsonify({'status': 'healthy', 'documents': len(api_doc_ids)})

def load_index_for_api():
    """Load the index when starting the API."""
    global api_vocabulary, api_tfidf_matrix, api_doc_ids
    with open('data/output/index.json', 'r', encoding='utf-8') as f:
        index = json.load(f)

    api_vocabulary = index['vocabulary']
    api_tfidf_matrix = np.array(index['tfidf_matrix'])
    api_doc_ids = index['document_ids']

    print(f"✓ API index loaded: {len(api_doc_ids)} documents, {len(api_vocabulary)}

def run_flask_in_background():
    """Run Flask in a background thread."""
    app.run(host='127.0.0.1', port=5000, debug=False, use_reloader=False)

# Load index and start server
load_index_for_api()

flask_thread = threading.Thread(target=run_flask_in_background, daemon=True)
flask_thread.start()

time.sleep(2)
```

```
✓ API index loaded: 3 documents, 4692 terms
 * Serving Flask app '__main__'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use
a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
```

Testing the Flask API on Command Prompt:

curl -X POST http://127.0.0.1:5000/search -H "Content-Type: application/json" -d "{"query": "information overload", "top_k": 3}"

Output:

{"query":"information overload","results":[{"document_id":"6B3BD97C-DEF2-49BB-B2B6-80F2CD53C4D3","rank":1,"score":0.36431398209036214},{"document_id":"0F64A61C-DF01-4F43-8B8D-F0319C41768E","rank":2,"score":0.07353215054370173},{"document_id":"1F648A7F-2C64-458C-BFAF-463A071530ED","rank":3,"score":0.06881616085499316}]}

Project Structure

```
    project/
    │
    ├── api/
    │   ├── __init__.py
```

```
│   ├── app.py
│   └── templates/
│       └── index.html
│
├── crawler/
│   ├── __pycache__/
│   ├── __init__.py
│   └── wiki_crawler.py
│
├── data/
│   ├── demo_corpus/
│   ├── html_corpus/
│   └── output/
│
├── indexer/
│   ├── __pycache__/
│   ├── __init__.py
│   ├── extractor.py
│   ├── indexer.py
│   └── utils.py
│
├── processor/
│   ├── __pycache__/
│   ├── __init__.py
│   ├── query_processor.py
│   ├── similarity.py
│   └── word2vec_search.py
│
├── Screenshot/
│
├── config.py
├── main.py
├── Notebook.ipynb
├── queries.csv
├── Readme.md
└── requirements.txt
```

GitHub Repo https://github.com/Theepankumargandhi/Information-Retrieval-Final_project