Information Retrieval Search Engine

Student: Aparnaa Mahalaxmi Arulljothi A20560995

---

1. **Crawler** – Scrapy tool used to gather a small set of HTML pages
2. **Indexer** – Builds a TF-IDF document index with scikit-learn
3. **Query Engine** – Ranks documents using cosine similarity

Information Retrieval Search Engine Project

- Scrapy-based Wikipedia crawl and TF-IDF index
- TF–IDF search over the given three HTML documents and a separately indexed Wikipedia corpus
- Query processing pipeline that writes ranked results to `results.csv`

In [1]:

```
from pathlib import Path
import os
import json
import csv
import re
import warnings
from bs4 import BeautifulSoup
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity as sklearn_cosine
import numpy as np
from flask import Flask, request, jsonify
from IPython.display import display, HTML
warnings.filterwarnings("ignore")
BASE_DIRS = [
    Path("data/wiki_corpus"),   # crawled Wikipedia pages
    Path("data/html_corpus"),   # given 3 HTML files
    Path("data/output"),        # index.json, wikipedia_index.json, results.csv
]
for path in BASE_DIRS:
    path.mkdir(parents=True, exist_ok=True)
    print(f"Directory ready: {path}")
```

```
Directory ready: data\wiki_corpus
Directory ready: data\html_corpus
Directory ready: data\output
```

Web Crawler

Scrapy spider used to collect a small Wikipedia corpus.

Config:

- Start URL: https://en.wikipedia.org/wiki/Information_retrieval
- Depth: up to 2 link levels
- Max pages: 100
- Saved as HTML under `data/wiki_corpus/`

HTML text extraction helper function

In [ ]:

```
def read_clean_html(path: Path) -> str:
    """Return plain text extracted from an HTML file."""
    try:
        with path.open("r", encoding="utf-8", errors="ignore") as f:
            html = f.read()
        soup = BeautifulSoup(html, "lxml")
        for tag in soup(["script", "style", "noscript", "meta", "header", "footer"]
            tag.decompose()
        for div in soup.find_all("div", {"class": ["mw-navigation", "vector-menu-co
            div.decompose()
        text = soup.get_text(" ", strip=True)  # Extract clean text
        text = re.sub(r"\s+", " ", text) # Remove multiple spaces
        return text
    except Exception as exc:
        print(f"Failed to read {path}: {exc}")
        return ""
```

Scrapy spider to grab up to 100 Wikipedia pages on Information Retrieval

In [3]:

```
import scrapy
from pathlib import Path
from scrapy.crawler import CrawlerProcess


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

    # Parameters
    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",
    }

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

    def parse(self, response):           # Extract the Wikipedia page title from th
        title = response.url.split("/wiki/")[-1]
        title = title.replace(" ", "_")         # Normalize any spaces
        filename = f"{title}.html"
        filepath = self.output_dir / filename
        filepath.write_bytes(response.body)
        self.logger.info(f"Saved {filepath} (Depth: {response.meta.get('depth')})")
        self.page_counter += 1
        if self.page_counter >= 100:                     # Stop if we have reached
            self.logger.info("Reached 100 pages. Stopping crawl.")
            return

        for href in response.css("a::attr(href)").getall():  # Follows internal Wik
            if href.startswith("/wiki/") and not href.startswith("/wiki/Special:")
                yield response.follow(href, callback=self.parse)

def run_crawler():
    print("Wikipedia Crawl Configuration")
    print("Max Depth     : 2")
    print("Page Limit    : 100\n")
    process = CrawlerProcess()
    process.crawl(WikipediaIR)
    process.start()
    print("\nCrawl completed.\n")


#run_crawler()
```

Preview of crawled Wikipedia pages

In [20]:

```
from pathlib import Path
print("Max Depth     : 1")
print("Page Limit    : 10\n")

wiki_dir = Path("data/wiki_corpus")
html_files = sorted(wiki_dir.glob("*.html"))

print(f"Total HTML files found: {len(html_files)}")

# cap at 100 pages
wiki_subset = html_files[:100]
print(f"Using {len(wiki_subset)} pages for indexing\n")

print("First 20 files:")
```

```
for f in wiki_subset[:20]:
    print(" ", f.name)

# a preview of the first crawled page
if wiki_subset:
    first_page = wiki_subset[0]
    print(f"\nPreview of: {first_page.name}")
    text = read_clean_html(first_page)

    print("\nExtracted text (first 500 chars):")
    print(text[:500])
else:
    print("No pages found in data/wiki_corpus/")
```

```
Max Depth     : 1
Page Limit    : 10

Total HTML files found: 100
Using 100 pages for indexing

First 20 files:
  1966_flood_of_the_Arno.html
  3D_retrieval.html
  BERT_(language_model).html
  Conservation-restoration_of_Leonardo_da_Vinci%27s_The_Last_Supper.html
  Conservation-restoration_of_the_H.L._Hunley.html
  Conservation-restoration_of_the_Shroud_of_Turin.html
  Conservation-restoration_of_the_Statue_of_Liberty.html
  Conservation-restoration_of_Thomas_Eakins%27_The_Gross_Clinic.html
  Conservation_and_restoration_of_Pompeian_frescoes.html
  Conservation_and_restoration_of_rail_vehicles.html
  Conservation_issues_of_Pompeii_and_Herculaneum.html
  Desktop_search.html
  Digital_libraries.html
  Ecce_Homo_(Garc%C3%ADa_Mart%C3%ADnez_and_Gim%C3%A9nez).html
  ElgooG.html
  Enterprise_search.html
  Ethnochoreology.html
  Ethnopoetics.html
  Family_folklore.html
  Federated_search.html

Preview of: 1966_flood_of_the_Arno.html

Extracted text (first 500 chars):
1966 flood of the Arno - Wikipedia Jump to content Contents move to sidebar hide (To
p) 1 Overview 2 Timeline of events Toggle Timeline of events subsection 2.1 3 Novemb
er 2.2 4 November 3 Impact Toggle Impact subsection 3.1 Collections affected 3.2 Wor
ks affected 4 Funding and assistance Toggle Funding and assistance subsection 4.1 Th
e "Mud Angels" 4.2 The "Flood Ladies" 5 Conservation measures Toggle Conservation me
asures subsection 5.1 Books and records 5.1.1 The National Library Centers of Fl
```

Load and inspect cleaned Wikipedia pages

In [5]:

```python
# Build a small doc dictionary from the crawled pages
wiki_docs = {}

print("\nLoading cleaned Wikipedia pages")
for html_file in wiki_subset:
    doc_id = html_file.stem
    text = read_clean_html(html_file)
    wiki_docs[doc_id] = text

print(f"Total documents loaded: {len(wiki_docs)}")
for i, (doc_id, text) in enumerate(wiki_docs.items()):  # few samples
    if i == 3:
        break
    print(f"Length: {len(text)} chars")
    print(f"Preview: {text[:200]}")
```

```
Loading cleaned Wikipedia pages
Total documents loaded: 100
Length: 31664 chars
Preview: 1966 flood of the Arno - Wikipedia Jump to content Contents move to sidebar
hide (Top) 1 Overview 2 Timeline of events Toggle Timeline of events subsection 2.1
3 November 2.2 4 November 3 Impact Toggl
Length: 7135 chars
Preview: 3D Content Retrieval - Wikipedia Jump to content Contents move to sidebar h
ide (Top) 1 3D retrieval methods 2 3D Engineering Search System 3 Challenges 4 See a
lso 5 References English Tools Tools move
Length: 46181 chars
Preview: BERT (language model) - Wikipedia Jump to content Contents move to sidebar
hide (Top) 1 Architecture Toggle Architecture subsection 1.1 Embedding 1.2 Architect
ural family 2 Training Toggle Training su
```

Build TF–IDF index for Wikipedia corpus

In [6]:

```python
# TF-IDF index for the crawled Wikipedia pages
print("\nBuilding TF-IDF index for Wikipedia corpus")

wiki_doc_ids = list(wiki_docs.keys())
wiki_texts = [wiki_docs[d] for d in wiki_doc_ids]

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

vectorizer_wiki = TfidfVectorizer(
    lowercase=True,
    stop_words="english",
    norm="l2",
)

wiki_tfidf = vectorizer_wiki.fit_transform(wiki_texts)
terms_wiki = vectorizer_wiki.get_feature_names_out()

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

sparsity = 1 - wiki_tfidf.nnz / (wiki_tfidf.shape[0] * wiki_tfidf.shape[1])
wikipedia_index = {
    "document_ids": wiki_doc_ids,
    "vocabulary": terms_wiki.tolist(),
    "tfidf_matrix": wiki_tfidf.toarray().tolist(),
    "vectorizer_params": {
        "lowercase": True,
        "stop_words": "english",
        "norm": "l2",
    },
}

wikipedia_index_path = Path("data/output/wikipedia_index.json")
wikipedia_index_path.parent.mkdir(parents=True, exist_ok=True)
with wikipedia_index_path.open("w", encoding="utf-8") as f:
    json.dump(wikipedia_index, f, indent=2)
print(f"Location: {wikipedia_index_path}")
print("Index summary:")
print(f"{len(wikipedia_index['document_ids'])} documents")
print(f"{len(wikipedia_index['vocabulary'])} terms")
```

```
Building TF-IDF index for Wikipedia corpus
Documents: 100
TF–IDF matrix created
Documents: 100
Vocabulary size: 35641 terms
Matrix shape: (100, 35641)
Location: data\output\wikipedia_index.json
Index summary:
100 documents
35641 terms
```

Inspect saved Wikipedia TF–IDF index

In [7]:

```
# Load wikipedia_index.json and print a quick summary
index_path = Path("data/output/wikipedia_index.json")
print("Loading Wikipedia TFIDF index from:", index_path)

with index_path.open("r", encoding="utf-8") as f:
    wikipedia_index = json.load(f)

print("\nIndex summary")
print("Documents:", len(wikipedia_index["document_ids"]))
print("Vocabulary size:", len(wikipedia_index["vocabulary"]))
print(
    "TFIDF matrix:",
    len(wikipedia_index["tfidf_matrix"]),
    "x",
    len(wikipedia_index["tfidf_matrix"][0]),
)

# a few document IDs
```

```python
print("\nSample document IDs:")
for doc_id in wikipedia_index["document_ids"][:5]:
    print("  ", doc_id)

# a few vocabulary terms
print("\nSample vocabulary terms:")
for term in wikipedia_index["vocabulary"][:15]:
    print("  ", term)

# first document vector preview
first_vec = wikipedia_index["tfidf_matrix"][0]
print("\nFirst document vector length:", len(first_vec))
print("First 20 TFIDF values:", first_vec[:20])
```

```
Loading Wikipedia TFIDF index from: data\output\wikipedia_index.json

Index summary
Documents: 100
Vocabulary size: 35641
TFIDF matrix: 100 x 35641

Sample document IDs:
   1966_flood_of_the_Arno
   3D_retrieval
   BERT_(language_model)
   Conservation-restoration_of_Leonardo_da_Vinci%27s_The_Last_Supper
   Conservation-restoration_of_the_H.L._Hunley

Sample vocabulary terms:
   00
   000
   0000
   00008
   0001
   0002
   00022
   000271620258300113
   00036
   0004
   00055
   00059
   0006
   0009
   000s

First document vector length: 35641
First 20 TFIDF values: [0.038543787214482, 0.0877668082367568, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.011105589430370672, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.
0]
```

In [ ]:

```python
import numpy as np
import json
from pathlib import Path

print("\nLoading Wikipedia TFIDF index")
index_path = Path("data/output/wikipedia_index.json")
```

```python
with index_path.open("r", encoding="utf-8") as f:
    wiki_index = json.load(f)

doc_ids = wiki_index["document_ids"]
vocab = wiki_index["vocabulary"]
tfidf_matrix = np.array(wiki_index["tfidf_matrix"])
print(f"Loaded index with {len(doc_ids)} documents and {len(vocab)} terms.\n")
vocab_index = {term: i for i, term in enumerate(vocab)}
def vectorize_query(query: str) -> np.ndarray: # Convert a query into a vector
    q_vec = np.zeros(len(vocab))
    for token in query.lower().split():
        if token in vocab_index:
            q_vec[vocab_index[token]] += 1.0   # simple weighted count
    return q_vec
def cosine_similarity(matrix, vector):
    vec_norm = np.linalg.norm(vector)
    doc_norms = np.linalg.norm(matrix, axis=1)
    sims = np.zeros(matrix.shape[0])
    valid = (vec_norm != 0) & (doc_norms != 0)
    sims[valid] = (matrix[valid] @ vector) / (doc_norms[valid] * vec_norm)
    return sims
query = "information retrieval system"
print(f"Query: \"{query}\"\n")

q_vec = vectorize_query(query)
scores = cosine_similarity(tfidf_matrix, q_vec)
top_k = 5  # Return top-5 results
ranked = scores.argsort()[::-1][:top_k]
print("Top matching Wikipedia documents:\n")
for rank, idx in enumerate(ranked, start=1):
    print(f"{rank}. {doc_ids[idx]:45s} similarity = {scores[idx]:.3f}")
```

```
Loading Wikipedia TFIDF index
Loaded index with 100 documents and 35641 terms.

Query: "information retrieval system"

Top matching Wikipedia documents:

1. Information_retrieval                         similarity = 0.714
2. Information_filtering                         similarity = 0.365
3. Music_information_retrieval                   similarity = 0.314
4. Image_retrieval                               similarity = 0.278
5. 3D_retrieval                                  similarity = 0.262
```

Document Indexer

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` Pipeline: HTML → cleaned text → TF–IDF → document–term matrix → `index.json`
  Output: `index.json` with document IDs, vocabulary, and TF–IDF weights for query processing

HTML parsing and clean text extraction (3 docs)

In [ ]:

```python
# Load 3 given  HTML documents and clean their text
official_files = [
    "0F64A61C-DF01-4F43-8B8D-F0319C41768E.html",
    "1F648A7F-2C64-458C-BFAF-463A071530ED.html",
    "6B3BD97C-DEF2-49BB-B2B6-80F2CD53C4D3.html",
]

corpus_dir = Path("data/html_corpus")
print("Checking official files...")

docs = {}
for fname in official_files:
    file_path = corpus_dir / fname
    if file_path.exists():
        doc_id = fname.replace(".html", "")
        text = read_clean_html(file_path)
        docs[doc_id] = text
        print(f"\nLoaded: {fname}")
        print(f"Text length: {len(text)} characters")
    else:
        print(f"\nMissing: {fname}")
print(f"\nTotal documents loaded: {len(docs)}")
```

```
Checking official files...

Loaded: 0F64A61C-DF01-4F43-8B8D-F0319C41768E.html
Text length: 56849 characters

Loaded: 1F648A7F-2C64-458C-BFAF-463A071530ED.html
Text length: 78758 characters

Loaded: 6B3BD97C-DEF2-49BB-B2B6-80F2CD53C4D3.html
Text length: 37277 characters

Total documents loaded: 3
```

Build TF–IDF vectors for the 3 given documents

In [ ]:

```python
# TF-IDF matrix for the three given docs
print("Building TFIDF index")

doc_ids = list(docs.keys())
doc_texts = [docs[d] for d in doc_ids]
```

```python
vectorizer = TfidfVectorizer(
    lowercase=True,
    stop_words="english",
    norm="l2",
)

tfidf_matrix = vectorizer.fit_transform(doc_texts)
feature_names = vectorizer.get_feature_names_out()
print("TFIDF 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}")
sparsity = 1 - tfidf_matrix.nnz / (tfidf_matrix.shape[0] * tfidf_matrix.shape[1])
```

```
Building TFIDF index
TFIDF matrix created
Documents: 3
Vocabulary size: 4544 unique terms
Matrix shape: (3, 4544)
```

Save TF–IDF index as index.json

In [ ]:

```python
# Create the JSON index
index_data = {
    "document_ids": doc_ids,
    "vocabulary": feature_names.tolist(),
    "tfidf_matrix": tfidf_matrix.toarray().tolist(),
    "vectorizer_params": {
        "lowercase": True,
        "stop_words": "english",
        "norm": "l2",
    },
}

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

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

print("Index saved successfully")
print("Location:", index_path)
print(f"File size: {os.path.getsize(index_path) / 1024:.2f} KB")

print("\nIndex Has:")
print(len(index_data["document_ids"]), "documents")
print( len(index_data["vocabulary"]), "vocabulary terms")
print(
    "TFIDF matrix:",
    len(index_data["tfidf_matrix"]),
    "x",
    len(index_data["tfidf_matrix"][0]),
)
```

```
Index saved successfully
Location: data\output\index.json
File size: 333.92 KB

Index Has:
3 documents
4544 vocabulary terms
TFIDF matrix: 3 x 4544
```

Part 3: Query Processor

Input: `index.json` (TF–IDF index) and `queries.csv` (query_id, query_text)
Pipeline: queries → TF–IDF → cosine similarity → ranked list → `results.csv`
Output: `results.csv` with columns: query_id, rank, document_id (all 3 docs ranked per query)

Load index.json and queries.csv

In [ ]:

```python
# Load TF-IDF index and queries
index_path = Path("data/output/index.json")
print("Loading index from:", index_path)

with index_path.open("r", encoding="utf-8") as f:
    idx = json.load(f)

doc_ids_loaded = idx["document_ids"]
vocab_loaded = idx["vocabulary"]
tfidf_loaded = np.array(idx["tfidf_matrix"])

print("\nIndex loaded")
print(f"Docs: {len(doc_ids_loaded)}")
print(f"Vocab size: {len(vocab_loaded)}")
print(f"Matrix: {tfidf_loaded.shape[0]} x {tfidf_loaded.shape[1]}")

# Load queries
queries = []
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("\nQueries loaded:", len(queries))

# preview
print("\nQuery preview:")
for q in queries:
    print(f"  {q['query_id']}: \"{q['query_text']}\"")
```

```
Loading index from: data\output\index.json

Index loaded
Docs: 3
Vocab size: 4544
Matrix: 3 x 4544

Queries loaded: 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"
```

Query processing function

In [ ]:

```python
def process_query(query_text, vocabulary, tfidf_matrix, doc_ids):
    """Run a query and return ranked docs."""
    # vectorize using the same vocab
    vec_q = TfidfVectorizer(
        lowercase=True,
        stop_words="english",
        vocabulary=vocabulary,
        norm="l2",
    )
    q_vec = vec_q.fit_transform([query_text]).toarray()
    sims = sklearn_cosine(q_vec, tfidf_matrix)[0]
    pairs = list(zip(doc_ids, sims))  # pair (doc_id, score) and sort
    pairs.sort(key=lambda x: x[1], reverse=True)
    ranked = [(i + 1, doc, score) for i, (doc, score) in enumerate(pairs)]  # add r
    return ranked

print("Query processor")
```

```
Query processor
```

Sample query analysis (query 1)

In [ ]:

```python
#  first query
print("Detailed query analysis\n")
sample = queries[0]
qid = sample["query_id"]
qtext = sample["query_text"]
print("Query ID:   ", qid)
print("Query text:", qtext)
print("Query terms:", qtext.lower().split())
# ranking
ranked = process_query(
    qtext,
    vocab_loaded,
    tfidf_loaded,
    doc_ids_loaded,
)
print("\nRanked documents:")
for rank, doc_id, score in ranked:
    print(f"  {rank}: {doc_id}   score={score:.6f}")
present_terms = [t for t in qtext.lower().split() if t in vocab_loaded] # which que
print("\nQuery terms found in vocabulary:", present_terms)
```

```
Detailed query analysis

Query ID:   6E93CDD1-52F9-4F41-A405-54E398EF6FF8
Query text: information overload
Query terms: ['information', 'overload']

Ranked documents:
  1: 6B3BD97C-DEF2-49BB-B2B6-80F2CD53C4D3   score=0.361912
  2: 0F64A61C-DF01-4F43-8B8D-F0319C41768E   score=0.072527
  3: 1F648A7F-2C64-458C-BFAF-463A071530ED   score=0.067744

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

Run all queries and save ranked results

In [ ]:

```
# process each query and collect all rankings
print("Running queries")
all_results = []
for q in queries:
    qid = q["query_id"]
    qtext = q["query_text"]
    print(f"\nQuery {qid}: \"{qtext}\"")
    ranked = process_query(
        qtext,
        vocab_loaded,
        tfidf_loaded,
        doc_ids_loaded,
    )
    for rank, doc_id, score in ranked:
        print(f"  {rank}. {doc_id}  ({score:.4f})")
        all_results.append({
            "query_id": qid,
            "rank": rank,
            "document_id": doc_id,
        })
out_path = Path("data/output/results.csv") # save results.csv
with out_path.open("w", newline="", encoding="utf-8") as f:
    writer = csv.DictWriter(f, fieldnames=["query_id", "rank", "document_id"])
    writer.writeheader()
    writer.writerows(all_results)
print("\nSaved results to:", out_path)
print("Total rows:", len(all_results))
```

```
Running queries

Query 6E93CDD1-52F9-4F41-A405-54E398EF6FF8: "information overload"
  1. 6B3BD97C-DEF2-49BB-B2B6-80F2CD53C4D3  (0.3619)
  2. 0F64A61C-DF01-4F43-8B8D-F0319C41768E  (0.0725)
  3. 1F648A7F-2C64-458C-BFAF-463A071530ED  (0.0677)

Query 0D97BCC6-C46E-4242-9777-7CEAED55B362: "database server hardware specs"
  1. 1F648A7F-2C64-458C-BFAF-463A071530ED  (0.3691)
  2. 0F64A61C-DF01-4F43-8B8D-F0319C41768E  (0.0227)
  3. 6B3BD97C-DEF2-49BB-B2B6-80F2CD53C4D3  (0.0158)

Query 78452FF4-94D7-422C-9283-A14615C44ADC: "search engine open sorce"
  1. 0F64A61C-DF01-4F43-8B8D-F0319C41768E  (0.5569)
  2. 6B3BD97C-DEF2-49BB-B2B6-80F2CD53C4D3  (0.1221)
  3. 1F648A7F-2C64-458C-BFAF-463A071530ED  (0.0287)

Saved results to: data\output\results.csv
Total rows: 9
```

Preview of generated results.csv

In [ ]:

```
# Read results.csv
results_path = Path("data/output/results.csv")
print("Loading results from:", results_path)

with results_path.open("r", encoding="utf-8") as f:
    reader = csv.DictReader(f)
    rows = list(reader)
print("\nFirst few rows:")
for row in rows[:10]:
    print(f"  {row['query_id']}  rank={row['rank']}  doc={row['document_id']}")

print("\nTotal rows:", len(rows))
```

```
Loading results from: data\output\results.csv

First few rows:
  6E93CDD1-52F9-4F41-A405-54E398EF6FF8  rank=1  doc=6B3BD97C-DEF2-49BB-B2B6-80F2CD53
C4D3
  6E93CDD1-52F9-4F41-A405-54E398EF6FF8  rank=2  doc=0F64A61C-DF01-4F43-8B8D-F0319C41
768E
  6E93CDD1-52F9-4F41-A405-54E398EF6FF8  rank=3  doc=1F648A7F-2C64-458C-BFAF-463A0715
30ED
  0D97BCC6-C46E-4242-9777-7CEAED55B362  rank=1  doc=1F648A7F-2C64-458C-BFAF-463A0715
30ED
  0D97BCC6-C46E-4242-9777-7CEAED55B362  rank=2  doc=0F64A61C-DF01-4F43-8B8D-F0319C41
768E
  0D97BCC6-C46E-4242-9777-7CEAED55B362  rank=3  doc=6B3BD97C-DEF2-49BB-B2B6-80F2CD53
C4D3
  78452FF4-94D7-422C-9283-A14615C44ADC  rank=1  doc=0F64A61C-DF01-4F43-8B8D-F0319C41
768E
  78452FF4-94D7-422C-9283-A14615C44ADC  rank=2  doc=6B3BD97C-DEF2-49BB-B2B6-80F2CD53
C4D3
  78452FF4-94D7-422C-9283-A14615C44ADC  rank=3  doc=1F648A7F-2C64-458C-BFAF-463A0715
30ED

Total rows: 9
```

Optional Enhancements

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

Optional: simple spell checker with NLTK

In [ ]:

```python
# NLTK-based spelling helper (for noisy queries)
import nltk
from nltk.corpus import words
from nltk.metrics import edit_distance
nltk.download("words", quiet=True)

english_vocab = set(words.words())
print("English word list loaded:", len(english_vocab))

def correct_spelling(word: str) -> str:
    """Return a simple spelling correction (or the word itself)."""
    w = word.lower()

    if w in english_vocab:  # already a known word
        return w

    # small pool of candidates with similar length
    cand = [t for t in english_vocab if abs(len(t) - len(w)) <= 2]
    if not cand:
        return w

    best = min(cand, key=lambda t: edit_distance(w, t))

    # only accept close matches
    if edit_distance(w, best) <= 2:
        return best
    return w

print("\nSpelling correction examples:")
for w in ["infomation", "Computr", "seaach", "retrieval"]:
    fixed = correct_spelling(w)
    status = "Changed" if fixed != w else "unchanged"
    print(f"  {w:12} {fixed:12}  ({status})")
```

```
English word list loaded: 235892

Spelling correction examples:
  infomation   infumation    (corrected)
  Computr      computer      (corrected)
  seaach       search        (corrected)
  retrieval    retrieval     (unchanged)
```

Part 4: Flask REST API

Simple REST interface for running searches.

Request: `POST /search` with `{"query": "text", "top_k": 3}`
Response: ranked documents and scores
Run: `python flask_app.py`

In [18]:

```
# Flask API
from flask import Flask, request, jsonify
import numpy as np
import json
from pathlib import Path

app = Flask(__name__)

# globals for the loaded index
vocab_api = None
matrix_api = None
docs_api = None


@app.route("/search", methods=["POST"])
def api_search():
    """Handle search requests and return ranked results."""
    try:
        body = request.get_json(silent=True)
        if not body:
            return jsonify({"error": "Missing JSON payload"}), 400
        query = body.get("query")
        if not query:
            return jsonify({"error": "Field 'query' is required"}), 400

        top_k = int(body.get("top_k", 3))

        if vocab_api is None or matrix_api is None or docs_api is None:
            return jsonify({"error": "Index not initialized"}), 503

        results = process_query(query, vocab_api, matrix_api, docs_api)

        output = [
            {
                "rank": r,
                "document_id": d,
                "score": float(s),
            }
            for r, d, s in results[:top_k]
        ]

        return jsonify({
            "query": query,
            "count": len(output),
            "results": output,
        }), 200

    except Exception as err:
        return jsonify({"error": str(err)}), 500


@app.route("/healthcheck", methods=["GET"])
def api_health():
    """Basic health and index status."""
    count = len(docs_api) if docs_api else 0
    return jsonify({
        "status": "running",
        "documents_loaded": count,
    }), 200


def load_index():
    """Load the index.json file into memory."""
    global vocab_api, matrix_api, docs_api

    idx_path = Path("data/output/index.json")
    with idx_path.open("r", encoding="utf-8") as f:
        data = json.load(f)

    docs_api = data["document_ids"]
    vocab_api = data["vocabulary"]
    matrix_api = np.array(data["tfidf_matrix"])

    print(f"[API] Loaded {len(docs_api)} docs and {len(vocab_api)} terms.")
```

```
# Run the API (use this when running as a script)
# if __name__ == "__main__":
#     load_index()
#     app.run(host="0.0.0.0", port=7000, debug=True)

print("Flask API ")
```

```
Flask API
```

In [ ]:

```
import threading
from flask import Flask, request, jsonify
import numpy as np
import json
from pathlib import Path
import time
app = Flask(__name__)
# Globals for the loaded index
vocab_api = None
matrix_api = None
docs_api = None

@app.route("/search", methods=["POST"])
def api_search():
    """Handle search requests and return ranked results."""
    try:
        body = request.get_json(silent=True)
        if not body:
            return jsonify({"error": "Missing JSON payload"}), 400

        query = body.get("query")
        if not query:
            return jsonify({"error": "Field 'query' is required"}), 400

        top_k = int(body.get("top_k", 3))

        if vocab_api is None or matrix_api is None or docs_api is None:
            return jsonify({"error": "Index not initialized"}), 503

        results = process_query(query, vocab_api, matrix_api, docs_api)

        output = [
            {
                "rank": r,
                "document_id": d,
                "score": float(s),
            }
            for r, d, s in results[:top_k]
        ]

        return jsonify({
            "query": query,
            "count": len(output),
            "results": output,
        }), 200

    except Exception as err:
```

```python
        return jsonify({"error": str(err)}), 500
@app.route("/health", methods=["GET"])
def api_health():
    """Basic health and index status."""
    count = len(docs_api) if docs_api else 0
    return jsonify({
        "status": "running",
        "documents_loaded": count,
    }), 200
@app.route("/")
def home():
    """Serve the web interface."""
    # just redirect or show a message
    return jsonify({
        "message": "Flask API is running!",
        "endpoints": {
            "/search": "POST - Search documents",
            "/health": "GET - Health check"
        }
    })

def load_index():
    """Load the index.json file into memory."""
    global vocab_api, matrix_api, docs_api

    idx_path = Path("data/output/index.json")
    with idx_path.open("r", encoding="utf-8") as f:
        data = json.load(f)

    docs_api = data["document_ids"]
    vocab_api = data["vocabulary"]
    matrix_api = np.array(data["tfidf_matrix"])

    print(f" API index loaded: {len(docs_api)} docs, {len(vocab_api)} terms")

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()
# Create and start background thread
flask_thread = threading.Thread(target=run_flask_in_background, daemon=True)
flask_thread.start()
time.sleep(2)
```

```
API index loaded: 3 docs, 4544 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 Windows (Command Prompt):

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

OUTPUT:

```
{"count":3,"query":"information retrieval","results":[{"document_id":"6B3BD97C-DEF2-49BB-B2B6-80F2CD53C4D3","rank":1,"score":0.7248218413342915},{"document_id":"0F64A61C-DF01-4F43-8B8D-F0319C41768E","rank":2,"score":0.0873194287706458},{"document_id":"1F648A7F-2C64-458C-BFAF-463A071530ED","rank":3,"score":0.08337342565124171}]}
```

https://github.com/aparnaa19/Final-IR-Project

Project Structure - Modularized version

```
project/
│
├── src/
│   ├── __init__.py
│   ├── crawler.py                    # Scrapy web crawler
│   ├── indexer.py                    # TF-IDF indexer
│   ├── query_processor.py            # Query ranking engine
│   └── utils.py                      # HTML parsing utilities
│
├── api/
│   ├── __init__.py
│   ├── app.py                        # Flask app
│   ├── static/                       # Static files
│   │   ├── css/
│   │   │   └── style.css
│   │   └── js/
│   │       └── script.js            # Frontend JS
│   └── templates/
│       └── index.html               # Frontend HTML
│
├── data/
│   ├── wiki_corpus/                 # Crawled Wikipedia pages
│   ├── html_corpus/                 # given 3 HTML files
│   │   ├── 0F64A61C-DF01-4F43-8B8D-F0319C41768E.html
│   │   ├── 1F648A7F-2C64-458C-BFAF-463A071530ED.html
│   │   └── 6B3BD97C-DEF2-49BB-B2B6-80F2CD53C4D3.html
│   └── output/
│       ├── index.json               # TF-IDF index
│       ├── results.csv              # Query results
│       └── wikipedia_index.json     # Wikipedia index
│
```

```
│
├── notebooks/
│   └── ir_system_report.ipynb        # Jupyter notebook
│
├── queries.csv                       # Input query file
├── requirements.txt
├── README.md
│── build_wiki_index.py
├── run_pipeline.py                   # Main pipeline script
└── test_system.py                    # Quick test script
```