Crawl_Search

December 4, 2025

Abstract

This project focuses on designing and implementing a web-based search pipeline, including document acquisition, indexing, and query retrieval. A Scrapy-powered crawler is used to download and store web documents in HTML format. From the HTML, I converted to TXT to generate an inverted index, calculated the TF-IDF weighting to support efficient similarity-based retrieval. Finally, a Flask-driven query processor enables users to submit free-text searches and receive ranked document results based on cosine similarity.

Future enhancements can focus on improving scalability and retrieval quality. Optional features such as concurrent and distributed crawling using AutoThrottle and Scrapyd would allow faster and broader document collection. Search accuracy could be increased by integrating semantic vector embeddings like Word2Vec or FAISS-based k-nearest neighbor similarity. Additionally, a front-end search interface and production deployment would make the system more user-accessible and robust.

Overview

This project centers on web document retrieval and query processing. It is built around three key components: web crawler, an indexer, and a query processor. Together, these components enable efficient crawling, indexing, and searching of web documents.

Solution Outline

1. **Web Crawler**: Uses Scrapy to crawl the web and download web documents.
2. **Indexer**: Builds an inverted index using TF-IDF and cosine similarity to support search and retrieval.
3. **Query Processor**: Accepts user queries and returns ranked document results using TF-IDF and cosine similarity.

Relevant Literature:

- Scrapy Documentation:
  This official tutorial provides a comprehensive introduction to Scrapy, a powerful Python framework for web crawling and scraping. It covers spider creation, data extraction, and best practices for scalable web data collection, which directly informs the design of the project's web crawler component.
- Flask Documentation:
  The Flask documentation offers detailed guidance on building lightweight web applications

and APIs in Python. It is the primary reference for implementing the query processor and REST API endpoints, enabling user interaction and search result delivery in this project.

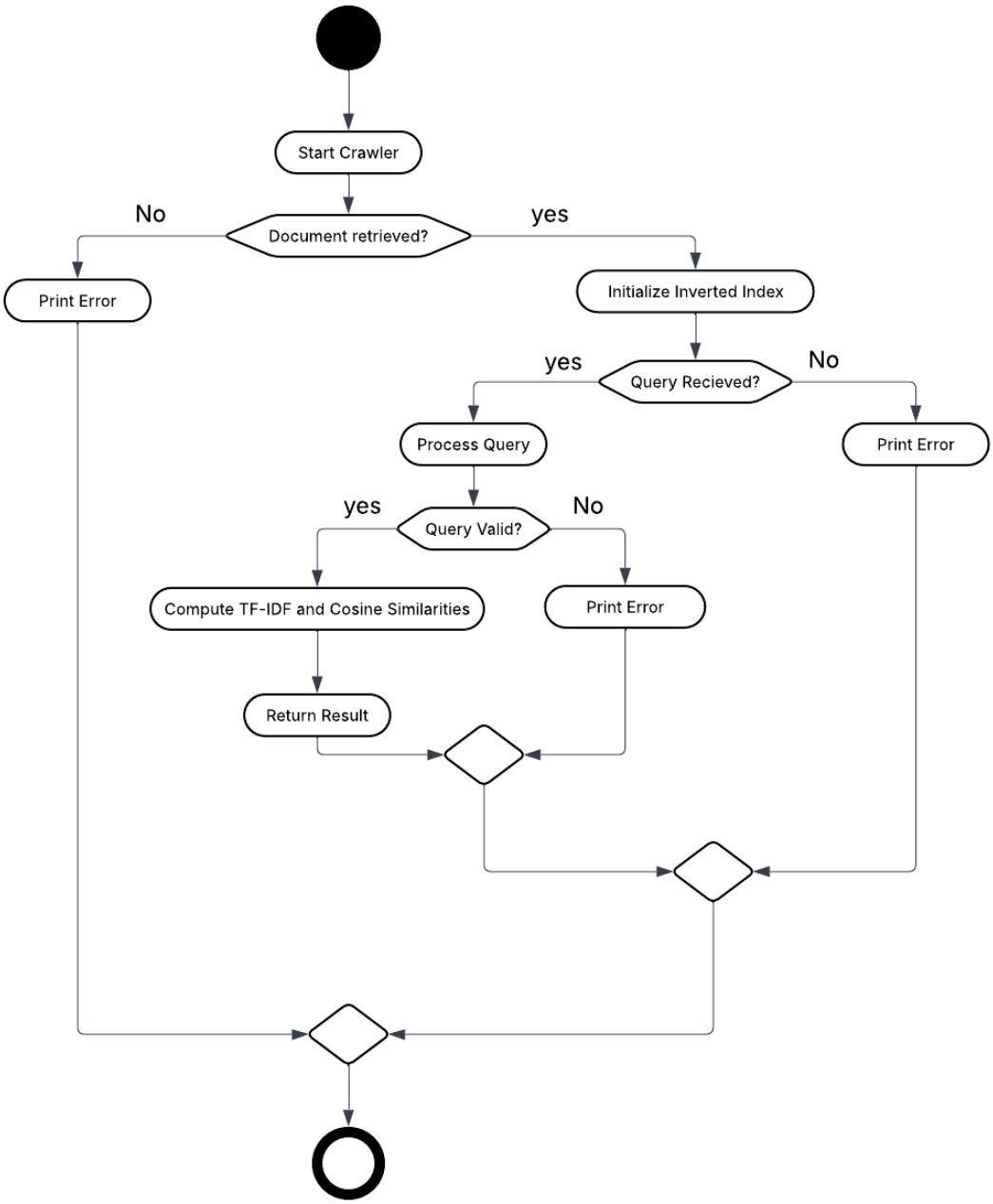
For the search indexing section, I have already studied and applied established techniques, as demonstrated in this project.

Proposed System

The system uses the power of Scrapy for web crawling, Scikit-Learn for calculating TF-IDF weights and cosine similarity, then Flask for query processing. The combination of these technologies delivers a solution for web document retrieval and query processing.

Design

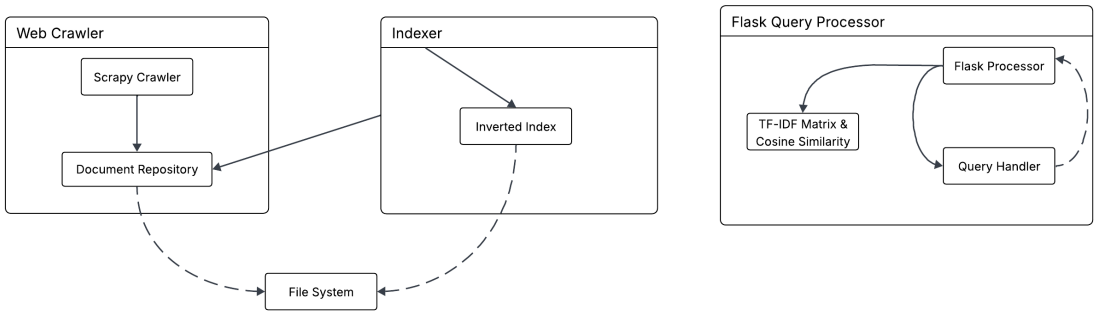
Processing user queries involves three integrated components: the Scrapy crawler, which collects web documents; the indexer, which constructs the TF-IDF matrix; and the Flask-based processor, which handles incoming queries. These components interact through structured data exchange and well-defined interfaces to ensure smooth integration across the system.



This activity diagram illustrates the sequence of operations within the system, including crawling, indexing, query processing, and delivering results. Each component works together to support the system's overall functionality.

Architecture

The system's architecture is composed of three core components: the Scrapy crawler, the inverted index generator, and the Flask-based query processor. Together, these components facilitate web document crawling, indexing, and query handling.



This diagram shows how the system's components interact and communicate. The system begin web scraping to retrieve HTML file, then convert the HTML files into TXT files. After the conversion, I generate the inverted index of the files. Once the inverted index has been created, the flask will then calculate the TF-IDF and cosine similularity based of the queries.csv file. Finally the system will output another CSV file named results of the ranking of each query based on the calculating the flask has made. The implementation leverages external libraries, including Scikit-Learn, BeautifulSoup, Scrapy, and Flask to provide the necessary functionality for crawling, parsing, indexing, and query processing.

Operation

Installation Instructions:
- Ensure Python 3.8 or higher is installed.
- Create and activate a virtual environment:
- `python -m venv venv`
- `venv\Scripts\activate` (Windows) or `source venv/bin/activate` (Linux/Mac)
- Install all required dependencies using:
- `pip install -r requirements.txt`

Software Commands:
- **Running the crawler:** Execute the relevant notebook cells or script to start the Scrapy crawler.
- **Building the index:** Run the indexing cells or script to generate the inverted index (`index.json`).
- **Running the Flask server and querying:** Start the Flask server by running the appropriate notebook cells or script. Submit queries via the web interface or batch CSV upload.

Inputs:
- **Seed URL:** The initial URL for the crawler.
- **CSV queries:** A CSV file containing `query_id` and `query_text` columns for batch search.
- **Configuration options:** Parameters such as maximum pages to crawl, crawl depth, and allowed domains.

Outputs:
- **Crawled HTML files:** Stored in the `pages/` directory.
- **Cleaned text files:** Stored in the `cleaned_text/` directory.
- **Inverted index:** Saved as `index.json`.
- **Search results:** Ranked results provided as a downloadable CSV file via the Flask API.

Conclusion

- **Success/Failure:**
  The project successfully demonstrates the implementation of a modular web search pipeline, encompassing web crawling, document cleaning, inverted indexing, and query processing. The system is capable of acquiring web documents, extracting and cleaning their content, constructing a positional inverted index, and retrieving relevant documents in response to user queries using TF-IDF and cosine similarity. Potential concerns include network interruptions, website changes, or anti-crawling measures that can disrupt document acquisition, as well as data quality issues or missing files that may affect indexing and search accuracy. Additionally, dependency mismatches, insufficient error handling, or scalability limitations could cause failures in processing or serving queries.
- **Outputs:**
  The outputs of the system are as follows:
  - Crawled HTML files are stored in the `pages/` directory.
  - Cleaned text files are saved in the `cleaned_text/` directory.
  - The positional inverted index is generated and saved as `index.json`.
  - Search results for user queries are provided as ranked CSV files, which can be downloaded via the Flask API.
  - Example outputs, including ranked document results and similarity scores for sample queries, are displayed below as evidence of successful retrieval and ranking.

Data Sources

- Web documents are sourced from quotes.toscrape.com
- Additional data sources can be integrated as needed for testing and evaluation.

Test Cases

Test cases focus on verifying the functionality of the crawler, indexer, and query processor. Tools such as Scrapy's built-in testing utilities and Python's unit testing frameworks can be used to support this process. Test coverage includes scenarios related to crawling behavior, index generation, and query handling.

Source Code

The complete source code for this project, including the web crawler, indexer, and query processor, is provided in the accompanying Jupyter notebook (`Crawl_Search.ipynb`) and related Python scripts within the project directory. All implementation details, code cells, and configuration files are included to enable full reproducibility of the results.

Key files and directories: - `Crawl_Search.ipynb`: Main notebook containing the code for crawling, indexing, and query processing. - `pages/`: Directory where crawled HTML files are stored. - `cleaned_text/`: Directory containing cleaned text files extracted from HTML pages. - `index.json`:

Generated inverted index used for search and retrieval. - `requirements.txt`: List of required Python packages for the project.

Bibliography

[1] Scrapy, "Scrapy Tutorial — Scrapy 2.3.0 documentation," 2012. [Online]. Available: https://docs.scrapy.org/en/latest/intro/tutorial.html

[2] L. Richardson, "Beautiful Soup Documentation — Beautiful Soup 4.4.0 Documentation," 2015. [Online]. Available: https://beautiful-soup-4.readthedocs.io/en/latest

[3] W3Schools, "Python JSON," 2025. [Online]. Available: https://www.w3schools.com/python/python_json.asp

[4] Scikit-learn, "sklearn.metrics.pairwise.cosine_similarity," 2018. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.pairwise.cosine_similarity.html

[5] Scikit-learn, "TfidfVectorizer," 2018. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html

[6] Flask, "Welcome to Flask — Flask Documentation (3.1.x)," 2025. [Online]. Available: https://flask.palletsprojects.com/en/stable

[2]:
```
import nest_asyncio
nest_asyncio.apply()

import scrapy
from scrapy.crawler import CrawlerProcess
from scrapy.linkextractors import LinkExtractor
import os
import uuid
```

Web Crawling

[ ]:
```
os.makedirs("pages", exist_ok=True)

class NotebookCrawler(scrapy.Spider):
    name = "notebook_crawler"

    def __init__(self, seed_url, allowed_domain, max_pages, max_depth, *args,␣
 ↪**kwargs):
        super(NotebookCrawler, self).__init__(*args, **kwargs)
        self.start_urls = [seed_url]
        self.allowed_domains = [allowed_domain]
        self.max_pages = max_pages
        self.visited = set()

        # Set max depth in custom settings
        self.custom_settings = {
            'DEPTH_LIMIT': max_depth,
```

```
            'AUTOTHROTTLE_ENABLED': True,
            'LOG_ENABLED': True,
            'CLOSESPIDER_PAGECOUNT': max_pages
        }

    def parse(self, response):
        page_uuid = str(uuid.uuid4()).upper() + ".html"

        with open(f"pages/{page_uuid}", "w", encoding="utf-8") as f:
            f.write(response.text)

        self.visited.add(response.url)

        # Continue crawling until max_pages reached
        if len(self.visited) < self.max_pages:
            links = LinkExtractor(allow_domains=self.allowed_domains).
 ↪extract_links(response)
            for link in links:
                if link.url not in self.visited:
                    yield response.follow(link.url, self.parse)

    # Only print once when crawler is done
    def closed(self, reason):
        self.logger.info(f"Total Documents Scraped: {len(self.visited)}")


# Crawler run
process = CrawlerProcess(settings={
    'LOG_LEVEL': 'INFO',
})

process.crawl(
    NotebookCrawler,
    seed_url='https://quotes.toscrape.com',
    allowed_domain='quotes.toscrape.com',
    max_pages=100,
    max_depth=5
)

try:
    process.start()
except:
    pass
```

```
2025-12-02 15:23:35 [scrapy.utils.log] INFO: Scrapy 2.13.3 started (bot:
scrapybot)
2025-12-02 15:23:35 [scrapy.utils.log] INFO: Versions:
{'lxml': '6.0.2',
```

```
 'libxml2': '2.11.9',
 'cssselect': '1.3.0',
 'parsel': '1.10.0',
 'w3lib': '2.3.1',
 'Twisted': '25.5.0',
 'Python': '3.14.0 (tags/v3.14.0:ebf955d, Oct  7 2025, 10:15:03) [MSC v.1944 '
           '64 bit (AMD64)]',
 'pyOpenSSL': '25.3.0 (OpenSSL 3.5.4 30 Sep 2025)',
 'cryptography': '46.0.3',
 'Platform': 'Windows-11-10.0.26200-SP0'}
2025-12-02 15:23:35 [scrapy.addons] INFO: Enabled addons:
[]
2025-12-02 15:23:35 [scrapy.extensions.telnet] INFO: Telnet Password:
30164f11bb523482
2025-12-02 15:23:35 [scrapy.middleware] INFO: Enabled extensions:
['scrapy.extensions.corestats.CoreStats',
 'scrapy.extensions.telnet.TelnetConsole',
 'scrapy.extensions.logstats.LogStats']
2025-12-02 15:23:35 [scrapy.crawler] INFO: Overridden settings:
{'LOG_LEVEL': 'INFO'}
2025-12-02 15:23:35 [scrapy.middleware] INFO: Enabled downloader middlewares:
['scrapy.downloadermiddlewares.offsite.OffsiteMiddleware',
 'scrapy.downloadermiddlewares.httpauth.HttpAuthMiddleware',
 'scrapy.downloadermiddlewares.downloadtimeout.DownloadTimeoutMiddleware',
 'scrapy.downloadermiddlewares.defaultheaders.DefaultHeadersMiddleware',
 'scrapy.downloadermiddlewares.useragent.UserAgentMiddleware',
 'scrapy.downloadermiddlewares.retry.RetryMiddleware',
 'scrapy.downloadermiddlewares.redirect.MetaRefreshMiddleware',
 'scrapy.downloadermiddlewares.httpcompression.HttpCompressionMiddleware',
 'scrapy.downloadermiddlewares.redirect.RedirectMiddleware',
 'scrapy.downloadermiddlewares.cookies.CookiesMiddleware',
 'scrapy.downloadermiddlewares.httpproxy.HttpProxyMiddleware',
 'scrapy.downloadermiddlewares.stats.DownloaderStats']
2025-12-02 15:23:35 [scrapy.middleware] INFO: Enabled spider middlewares:
['scrapy.spidermiddlewares.start.StartSpiderMiddleware',
 'scrapy.spidermiddlewares.httperror.HttpErrorMiddleware',
 'scrapy.spidermiddlewares.referer.RefererMiddleware',
 'scrapy.spidermiddlewares.urllength.UrlLengthMiddleware',
 'scrapy.spidermiddlewares.depth.DepthMiddleware']
2025-12-02 15:23:35 [scrapy.middleware] INFO: Enabled item pipelines:
[]

2025-12-02 15:23:35 [scrapy.core.engine] INFO: Spider opened
2025-12-02 15:23:35 [scrapy.extensions.logstats] INFO: Crawled 0 pages (at 0
pages/min), scraped 0 items (at 0 items/min)
2025-12-02 15:23:35 [scrapy.extensions.telnet] INFO: Telnet console listening on
127.0.0.1:6023
2025-12-02 15:23:38 [scrapy.core.engine] INFO: Closing spider (finished)
```

```
2025-12-02 15:23:38 [notebook_crawler] INFO: Total Documents Scraped: 254
2025-12-02 15:23:38 [scrapy.statscollectors] INFO: Dumping Scrapy stats:
{'downloader/request_bytes': 129050,
 'downloader/request_count': 323,
 'downloader/request_method_count/GET': 323,
 'downloader/response_bytes': 1361266,
 'downloader/response_count': 323,
 'downloader/response_status_count/200': 254,
 'downloader/response_status_count/308': 69,
 'dupefilter/filtered': 877,
 'elapsed_time_seconds': 3.771481,
 'finish_reason': 'finished',
 'finish_time': datetime.datetime(2025, 12, 2, 21, 23, 38, 969220,
tzinfo=datetime.timezone.utc),
 'items_per_minute': 0.0,
 'log_count/INFO': 11,
 'request_depth_max': 5,
 'response_received_count': 254,
 'responses_per_minute': 5080.0,
 'scheduler/dequeued': 323,
 'scheduler/dequeued/memory': 323,
 'scheduler/enqueued': 323,
 'scheduler/enqueued/memory': 323,
 'start_time': datetime.datetime(2025, 12, 2, 21, 23, 35, 197739,
tzinfo=datetime.timezone.utc)}
2025-12-02 15:23:38 [scrapy.core.engine] INFO: Spider closed (finished)
```

Indexing

[ ]:
```python
from bs4 import BeautifulSoup
import os

# Create folder for cleaned text files
os.makedirs("cleaned_text", exist_ok=True)

def html_to_text(html_file):
    with open(html_file, 'r', encoding='utf-8') as f:
        soup = BeautifulSoup(f, 'html.parser')

    for script in soup(["script", "style"]):
        script.decompose()

    text = soup.get_text()

    lines = (line.strip() for line in text.splitlines())
    chunks = (phrase.strip() for line in lines for phrase in line.split("  "))
    text = ' '.join(chunk for chunk in chunks if chunk)
```

```
    return text

pages_dir = "pages"
documents = {}
count = 0

for filename in os.listdir(pages_dir):
    if filename.endswith('.html'):
        filepath = os.path.join(pages_dir, filename)
        clean_text = html_to_text(filepath)
        documents[filename] = clean_text

        text_filename = filename.replace('.html', '.txt')
        with open(f"cleaned_text/{text_filename}", 'w', encoding='utf-8') as f:
            f.write(clean_text)

        count += 1

print(f"\nSuccessfully processed {count} documents into cleaned_text folder.")
```

```
Successfully processed 254 documents into cleaned_text folder.
```

[6]:
```
import json
import re
from collections import defaultdict

def tokenize_with_positions(text):
    text = text.lower()
    tokens = re.findall(r'\b[a-z]+\b', text)

    token_positions = defaultdict(list)
    for pos, token in enumerate(tokens):
        token_positions[token].append(pos)

    return dict(token_positions)

def build_positional_inverted_index(documents):
    inverted_index = defaultdict(list)

    for doc_id, text in documents.items():
        clean_doc_id = doc_id.replace('.html', '')

        token_positions = tokenize_with_positions(text)

        for token, positions in token_positions.items():
```

```
            inverted_index[token].append([clean_doc_id, positions])

    return dict(inverted_index)

inverted_index = build_positional_inverted_index(documents)

# Save to JSON file with custom formatting
with open('index.json', 'w', encoding='utf-8') as f:
    f.write('{\n')
    items = list(inverted_index.items())
    for i, (token, entries) in enumerate(items):
        f.write(f'  "{token}": [\n')
        for j, entry in enumerate(entries):
            entry_json = json.dumps(entry)
            if j < len(entries) - 1:
                f.write(f'    {entry_json},\n')
            else:
                f.write(f'    {entry_json}\n')
        if i < len(items) - 1:
            f.write('  ],\n')
        else:
            f.write('  ]\n')
    f.write('}\n')

print(f"Inverted index saved to index.json")
print(f"Total unique tokens: {len(inverted_index)}")
```

```
Inverted index saved to index.json
Total unique tokens: 3983
```

[7]:
```
import json
import os
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np

# Load the positional inverted index from index.json
with open('index.json', 'r', encoding='utf-8') as f:
    index = json.load(f)

# Get all doc_ids from the index
all_doc_ids = set()
for token_entries in index.values():
    for doc_id, _ in token_entries:
        all_doc_ids.add(doc_id)
all_doc_ids = list(all_doc_ids)
```

```
def get_doc_text(doc_id):
    filepath = f"cleaned_text/{doc_id}.txt"
    if os.path.exists(filepath):
        with open(filepath, 'r', encoding='utf-8') as f:
            return f.read()
    return ""

def search_tfidf(query_text, top_k):
    # Use only doc_ids from index.json
    doc_id_list = sorted(all_doc_ids)
    corpus = [get_doc_text(doc_id) for doc_id in doc_id_list]
    vectorizer = TfidfVectorizer(lowercase=True, token_pattern=r'\b[a-z]+\b')
    tfidf_matrix = vectorizer.fit_transform(corpus)
    feature_names = vectorizer.get_feature_names_out()
    query_terms = [t for t in query_text.lower().split() if t.isalpha() or t.
 ↪isalnum()]
    query_string = ' '.join(query_terms)
    query_vector = vectorizer.transform([query_string])
    similarities = cosine_similarity(query_vector, tfidf_matrix).flatten()
    results = []
    for idx, doc_id in enumerate(doc_id_list):
        score = similarities[idx]
        tfidf_weights = {}
        doc_vector = tfidf_matrix[idx]
        for query_term in query_terms:
            if query_term in feature_names:
                term_idx = np.where(feature_names == query_term)[0]
                tfidf_weights[query_term] = doc_vector[0, term_idx[0]] if␣
 ↪len(term_idx) > 0 else 0.0
            else:
                tfidf_weights[query_term] = 0.0
        results.append((doc_id, score, tfidf_weights))
    results.sort(key=lambda x: x[1], reverse=True)
    return results[:top_k]

# Example query
query = "better to be"
results = search_tfidf(query, top_k=5)
print(f"\nTop {5} results for query: {query}\n")
for rank, (doc_id, score, tfidf_weights) in enumerate(results, 1):
    print(f"{rank}. Document: {doc_id}")
    print(f"   Cosine Similarity Score: {score:.4f}")
    print(f"   TF-IDF Weights: {', '.join([f'{term}: {weight:.4f}' for term,␣
 ↪weight in tfidf_weights.items()])}")
    print()
```

```
Top 5 results for query: better to be

1. Document: 4E638282-3800-49BA-A2C5-4E9BD45850AA
   Cosine Similarity Score: 0.3424
   TF-IDF Weights: better: 0.1825, to: 0.1587, be: 0.3121

2. Document: EAAAA32D-17A2-442E-B578-1F748045BCA2
   Cosine Similarity Score: 0.3424
   TF-IDF Weights: better: 0.1825, to: 0.1587, be: 0.3121

3. Document: 2A38CD67-658C-46EB-B447-BECD2960EC47
   Cosine Similarity Score: 0.2161
   TF-IDF Weights: better: 0.1055, to: 0.1071, be: 0.2105

4. Document: 53477293-9075-4EF3-B04C-197B325F5411
   Cosine Similarity Score: 0.2161
   TF-IDF Weights: better: 0.1055, to: 0.1071, be: 0.2105

5. Document: C057C704-F881-4B6C-9379-124C10BC642A
   Cosine Similarity Score: 0.2161
   TF-IDF Weights: better: 0.1055, to: 0.1071, be: 0.2105
```

Query Processing

[ ]:
```python
import csv
import uuid
import random

sample_queries = [
    "to be or not to be",
    "We read to know we're not alone.",
    "cup of tea large enough"
]

with open('queries.csv', 'w', newline='', encoding='utf-8') as f:
    writer = csv.writer(f)
    writer.writerow(['query_id', 'query_text'])
    for q in sample_queries:
        writer.writerow([str(uuid.uuid4()).upper(), q])
print("Sample queries.csv created.")

with open('queries.csv', 'r', encoding='utf-8') as f:
    reader = csv.DictReader(f)
    queries = list(reader)
```

```
Sample queries.csv created.
```

[8]:
```
from flask import Flask, request, jsonify, send_file
import os
import csv
import io
import json
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
import re

app = Flask(__name__)

# Global variables
all_doc_ids = None
inverted_index = None


def load_index_metadata(index_path='index.json'):
    """
    Load index.json to get all document IDs
    """
    global all_doc_ids, inverted_index

    print(f"Loading index from {index_path}...")

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

    # Extract all unique document IDs from the index
    all_doc_ids = set()
    for term, postings in inverted_index.items():
        for doc_id, positions in postings:
            all_doc_ids.add(doc_id)

    all_doc_ids = sorted(list(all_doc_ids))

    print(f"  Index loaded: {len(inverted_index)} terms, {len(all_doc_ids)}␣
 ↪documents")

    return all_doc_ids


def get_doc_text(doc_id):
    """
    Load document text from cleaned_text directory
    """
    file_path = os.path.join('cleaned_text', f'{doc_id}.txt')
```

```
    if not os.path.exists(file_path):
        return ""
    with open(file_path, 'r', encoding='utf-8') as f:
        return f.read()


def search_tfidf(query_text, top_k):
    doc_id_list = sorted(all_doc_ids)
    corpus = [get_doc_text(doc_id) for doc_id in doc_id_list]

    vectorizer = TfidfVectorizer(lowercase=True, token_pattern=r'\b[a-z]+\b')
    tfidf_matrix = vectorizer.fit_transform(corpus)

    query_vector = vectorizer.transform([query_text.lower()])
    similarities = cosine_similarity(query_vector, tfidf_matrix).flatten()

    ranked_results = list(zip(doc_id_list, similarities))
    ranked_results.sort(key=lambda x: x[1], reverse=True)

    return ranked_results[:top_k]


def validate_query(query_text):
    if not query_text or not isinstance(query_text, str):
        return False, None, "Query text is required"
    query_text = query_text.strip()
    if len(query_text) == 0:
        return False, None, "Query cannot be empty"
    if len(query_text) > 500:
        return False, None, "Query too long"
    return True, query_text, None


def parse_query_csv(file_content):
    queries = []
    csv_data = io.StringIO(file_content.decode('utf-8'))
    reader = csv.DictReader(csv_data)

    if 'query_id' not in reader.fieldnames or 'query_text' not in reader.
 ↪fieldnames:
        raise ValueError("CSV must contain query_id and query_text")

    for row in reader:
        if not row['query_id'] or not row['query_text']:
            continue
        queries.append((row['query_id'].strip(), row['query_text'].strip()))
```

```
    return queries


@app.route('/')
def home():
    return jsonify({
        "message": "Search Engine Query Processor",
        "documents_loaded": len(all_doc_ids) if all_doc_ids else 0,
        "endpoints": {
            "/search/batch": "Upload CSV → download ranked CSV",
        }
    })


@app.route('/search/batch', methods=['POST'])
def search_batch_endpoint():
    if all_doc_ids is None:
        return jsonify({"error": "Index not loaded"}), 500

    if 'file' not in request.files:
        return jsonify({"error": "Missing CSV file"}), 400

    file = request.files['file']
    top_k = int(request.form.get('top_k', 3))
    csv_content = file.read()

    queries = parse_query_csv(csv_content)
    results_out = []

    for query_id, text in queries:
        valid, cleaned, err = validate_query(text)
        if not valid:
            continue

        results = search_tfidf(cleaned, top_k)

        for rank, (doc_id, score) in enumerate(results, 1):
            results_out.append({
                "query_id": query_id,
                "rank": rank,
                "document_id": doc_id
            })

    # Create output CSV
    output = io.StringIO()
    writer = csv.DictWriter(output, fieldnames=["query_id", "rank",␣
 ↪"document_id"])
```

```
    writer.writeheader()
    writer.writerows(results_out)
    output.seek(0)

    return send_file(
        io.BytesIO(output.getvalue().encode('utf-8')),
        mimetype='text/csv',
        as_attachment=True,
        download_name="results.csv"
    )


# --------------------------
# Start server
# --------------------------
if __name__ == '__main__':
    if not os.path.exists('index.json'):
        print("ERROR: index.json not found!")
    else:
        load_index_metadata()

    if not os.path.exists('cleaned_text'):
        print("WARNING: cleaned_text directory missing!")

    app.run(debug=True, host='0.0.0.0', port=5000, use_reloader=False)
```

```
Loading index from index.json…
  Index loaded: 3983 terms, 254 documents
 * Serving Flask app '__main__'
 * Debug mode: on
2025-12-02 15:26:48 [werkzeug] INFO: WARNING: This is a development
server. Do not use it in a production deployment. Use a production WSGI server
instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5000
 * Running on http://104.194.117.44:5000
2025-12-02 15:26:48 [werkzeug] INFO: Press CTRL+C to quit
2025-12-02 15:26:54 [werkzeug] INFO: 127.0.0.1 - - [02/Dec/2025 15:26:54] "POST
/search/batch HTTP/1.1" 200 -
```