Project Report: Searchme

Wikrama W. Wardhana, Illinois Institute of Technology, wwardhana@hawk.illinoistech.edu

Abstract

This report presents Searchme, a web-based information retrieval system implementing the vector space model with TF-IDF term weighting and cosine similarity ranking. The system comprises three integrated components: a Scrapy-based web crawler configured for breadth-first traversal with politeness controls, a multithreaded indexer that constructs inverted indices with positional information and bigram character indices for spelling correction, and a query processor that handles natural language queries through tokenization, correction, and ranked retrieval. The indexer employs magnitude caching to optimize repeated cosine similarity calculations, while the bigram index enables approximate string matching for handling misspelled queries. Searchme was evaluated on the Cranfield benchmark dataset of 1,400 documents and 225 queries, achieving a Mean Average Precision of 0.27, Precision@10 of 0.22, and Recall@100 of 0.70. These results passes simple baseline systems but fall short of modern ranking functions like BM25.

Overview

The objective of this project is to create an information retrieval system akin to a search engine. To mimic this behaviour, several key characteristics are required to consider the implementation as a derivative from such systems. The three core components this system employs: a Scrapy-based web crawler for document collection, a multithreaded indexer that builds tf-idf weighted inverted indices, and a query processor that ranks documents using cosine similarity. The system was evaluated against the Cranfield dataset to measure retrieval effectiveness using standard IR metrics.

Document Corpus Collection

For an information retrieval system to function, it must first have access to the corpus containing said information. In the context of search engines, the system must therefore have a collection of website information and its content stored and preprocessed for extraction; the relevant information is then transformed into its appropriate data structures for the system. This was achieved through the use of Scrapy's web crawler to collect and download web pages for indexing. The crawler accepts a seed URL along with preconfigured parameters for maximum crawl depth and page limits, ensuring all documents are stored locally in HTML format. This approach allows the system to build a custom corpus tailored to specific topics. The crawler generates both the raw HTML files and a mapping file

(url_map.jsonl) that associates document identifiers with their original URLs. Implementation details, including crawling strategies and storage organization, are discussed in the Architecture section.

Corpus Processing and Inverted Index Creation

Once an information retrieval system has access to its corpus of documents, its next step is to process its documents along with the content into the relevant data structures that builds the core of the entire system. In this system, the "Indexer" serves that exact purpose: it transforms raw HTML documents into searchable data structures. It parses HTML content using BeautifulSoup, extracts only the texts, and tokenizes it while filtering stopwords and punctuation. The core data structure is an inverted index that maps terms to document identifiers and positional information, represented using tf-idf weights. To improve performance, the indexer employs multithreaded processing that distributes tokenization across CPU cores. Key design decisions regarding tokenization strategies, threading implementation, and index persistence are detailed in the Architecture section.

Query Processing and Document Retrieval

The last key component is the user-facing interface that handles input queries and perform the necessary preprocessing before executing the information retrieval pipeline. It accepts natural language queries, applies the same preprocessing pipeline used during indexing (tokenization, stopword removal, case normalization), and generates a query vector using tf-idf weighting. The system then computes cosine similarity scores between the query vector and all candidate documents, ranking them by relevance. To manage computational load and improve user experience, Searchme implements top-K retrieval, returning only the most relevant documents. The system also accommodates user input variations through its bigram index, which supports wildcard matching for misspelled or partially-known terms. The Design section outlines the complete information retrieval pipeline from query input to ranked result presentation, while implementation specifics are provided in the Architecture section.

Design

This section describes the overarching design of Searchme's information retrieval system, describing its system capabilities, component interactions, and key design decisions that shaped the implementation.

System Capabilities

Searchme provides several core capabilities that enable effective information retrieval from a collection of web documents:

Index Construction

The system processes HTML documents to build an inverted index with tf-idf weighted term representations. Each term in the index maintains positional information, recording the exact locations where terms appear within documents to enable future extensions, such as phrase queries or proximity-based ranking.

Query Processing

Users can submit queries that are tokenized and processed using the same preprocessing pipeline applied during indexing. The system converts queries into tf-idf weighted vectors and computes cosine similarity scores against all indexed documents to determine relevance.

Ranked Retrieval

Rather than returning all matching documents, Searchme implements top-K retrieval to present only the most relevant results calculated by their cosine similarity scores.

Wildcard Query Support

Through a bigram character index, the system supports wildcard queries that can match terms with partial or uncertain spellings. This accommodates user input variations and helps retrieve relevant documents even when exact term matches are not found.

Efficient Processing

The system uses multithreaded document processing to leverage multiple CPU cores during index construction, and caches document magnitude calculations to avoid redundant computations and speed up query evaluation.

Component Interactions and Data Flow

The web crawler serves as the entry point to the system. Given the preconfigured parameters of a seed URL, maximum depth, and page limit, it traverses the web by following hyperlinks and downloading HTML content. The crawler outputs two artifacts: a directory of HTML files named by unique document identifiers, and a JSON Lines file that preserves the its document IDs and original URLs mapping. This separation allows the indexer to process documents efficiently while maintaining traceability to source locations for later components to use.

The indexer reads HTML files from the corpus directory and transforms them into searchable data structures. Each document passes through a processing pipeline that extracts text content using HTML parsing, converts text to lowercase for case-insensitive matching, removes punctuation through regular expression substitution, and filters common stopwords that provide little informational value. The resulting tokens are organized into an inverted index where each term maps to a posting list containing document IDs and their respective positions for when it appears. Simultaneously, a bigram index is constructed by generating character bigrams for each term, enabling wildcard matching capabilities. The complete

index structure, along with corpus metadata, is serialized to JSON format for persistence and future loading.

When a user submits a query, the processor applies the same tokenization and preprocessing steps used during indexing to ensure consistent term representation. The processed query tokens are converted into a tf-idf weighted vector by computing term frequencies within the query and multiplying by inverse document frequency values from the index. The system then iterates through the inverted index, computing dot products between the query vector and document vectors for all candidate documents containing at least one query term. These raw dot product scores are normalized by the magnitudes of both the query vector and each document vector to produce cosine similarity scores bounded between zero and one. Finally, documents are sorted by score in descending order, and the top-K results are returned to the user.

Integration

The three components are loosely coupled through file system conventions and data formats. The crawler and indexer communicate through a shared directory structure where HTML files and the URL mapping file reside. The indexer and query processor share access to the serialized index file, which contains all necessary data structures for retrieval. This file-based integration approach provides several advantages: components can be developed and tested independently, the system naturally supports batch processing where corpus collection and indexing occur offline while query processing happens online, and the serialized index serves as a checkpoint that eliminates the need to rebuild indices for repeated query sessions. The indexer implements lazy loading, creating the index only when no existing index file is found. This design allows the system to skip expensive index construction when working with a previously processed corpus, reducing startup time for query processing tasks.

Architecture

This following section will detail code implementations across the three core components, pulling snippets from the source code, noting key design decisions and its implications, and how it integrates with the entire system.

Web crawler

```
custom_settings: dict[str, Any] = {
    "DEPTH_LIMIT": 3,
    "CLOSESPIDER_PAGECOUNT": 5000,
    "AUTOTHROTTLE_ENABLED": True,
    "AUTOTHROTTLE_START_DELAY": 1,
    "AUTOTHROTTLE_MAX_DELAY": 5,
    "AUTOTHROTTLE_TARGET_CONCURRENCY": 1.5,
```

```
    "CONCURRENT_REQUESTS": 16,
    "CONCURRENT_REQUESTS_PER_DOMAIN": 2,
    "ROBOTSTXT_OBEY": True,
}
```

The web crawler is preconfigured with values that best support the functions of a search engine. Namely, Searchme settled on using a small `DEPTH_LIMIT` of 3 combined with a significantly larger `CLOSESPIDER_PAGECOUNT` of 5000. This configuration encourages breadth-first traversal, where the crawler explores many pages at shallow depths rather than following deeply nested link chains. This approach captures diverse content across the crawled site(s) without getting trapped in deep hierarchical structures or infinite link loops. And to assist the crawling process, several autothrottling and concurrent settings are configured as to better optimize the process and maintain politeness of target websites so as to not overload them with requests.

```
@staticmethod
def only_http_https(url: str):
    scheme = urlparse(url).scheme.lower()
    if scheme in {"", "http", "https"}:
        return url
    return None

link_extractor = LinkExtractor(
    process_value=only_http_https,
    allow_domains=["en.wikipedia.org"],
    deny=[
        r"/user/",
        r"/profile/",
        r"/account/",
        r"/login",
        r"/register",
        r"/signup",
        r"\?action=",
        r"/edit",
        r"/delete",
        r"/admin",
        r"/tag/",
        r"/category/",
        r"/archive/",
        r"/search",
        r"/share",
        r"/print",
        r"/comment",
        r"#comment",
    ],
    deny_extensions=["pdf", "zip", "gz", "tar", "7z", "rar"],
    tags=["a"],
    attrs=["href"],
    canonicalize=True,
    unique=True,
)
```

To better control the behaviour of Searchme's crawler, the spider utilizes Scrapy's `LinkExtractor` object to parse through hyperlinks and filter out candidate hops between pages. For the purposes of this system, it is limited within the domain of `en.wikipedia.org` as a promising seeding URL resulting in a sufficient corpus to test robustness of the entire system. Morever, the link extractor employs further URL parsing to ensure only HTTP/HTTPS web pages are crawled, ignoring common page links that serve little to no informational gain, files nested for download within the pages themselves that is not supported by this system, and strictly download a specific URL once to ensure no duplicates are downloaded.

```
@classmethod
def from_crawler(cls, crawler, *args, **kwargs):
    output_path_arg = kwargs.get("output_path", cls.default_output_path)
    output_path = Path(output_path_arg)

    feeds = {
        str(output_path / "url_map.jsonl"): {"format": "jsonlines"},
    }
    crawler.settings.set("FEEDS", feeds, priority="spider")

    spider = super().from_crawler(crawler, *args, **kwargs)
    return spider
```

The spider is then configured to maintain a trace of all downloaded HTML pages by creating a JSON line file that maps their respective URL and the parsed document ID, performed by the parser of the same spider.

```
def parse(self, response):
        self.log(f"Scraped URL @ {response.url}")
        docID = str(uuid.uuid5(uuid.NAMESPACE_URL, response.url))

        # save for mapping (goes into url_map.jsonl)
        yield {"docID": docID, "url": response.url}

        html_output_path = self.output_path / "html"
        filename = f"{docID}.html"

        html_output_path.mkdir(parents=True, exist_ok=True)
        (html_output_path / filename).write_bytes(response.body)
        self.log(f"Saved file {filename}")

        links = self.link_extractor.extract_links(response)
        yield from response.follow_all(links, callback=self.parse)
```

Finally, the parser ties the previous together. All scraped URLs are transformed into a unique UUID as their document ID representaion, and its mapping is then written into the aforementioned JSON line file. Then, all scraped hyperlink elements from the downloaded HTML are passed through the link extractor object previously defined to be filtered so as to minimize bad crawls, and the spider will recursively traverse each links until it reaches the page count limits.

Indexer

```
def init_invidx_tfdf(doc_idx: dict[str, list[str]]):
    inverted_idx: dict[str, dict[str, list[int]]] = {}

    for doc, term_lst in doc_idx.items():

        # maintain list of positions of term per document
        # note it is always sorted as it assumes list of text is in
original ordering
        for pos, term in enumerate(term_lst):
            if term in inverted_idx:
                posting_list = inverted_idx[term].setdefault(doc, [])
                posting_list.append(pos)
            else:
                inverted_idx[term] = {doc: [pos]}

    return inverted_idx
```

The core of Searchme's indexer lies in the inverted index creation function. Simply put, given a dictionary of documet IDs and their respective list of tokens, the function iterates through each document and created the inverted index consisting of unique terms while noting each documents containing said terms. Moreover, each position of the terms are recorded for each document to enable future expansion for positional information processing when querying.

```
# Attr:
# https://stackoverflow.com/questions/312443/how-do-i-split-a-list-into-
equally-sized-chunks
def chunks(lst, n):
    """Yield successive n-sized chunks from lst."""
    for i in range(0, len(lst), n):
        yield lst[i : i + n]


def html_tokenizer(html_path: str):
    p = Path(html_path)

    with open(p, "rb") as html:
        soup_obj = BeautifulSoup(html, "html.parser")
        content = soup_obj.get_text(" ", strip=True).lower()

    regex_p = r"[^\w\s]"
    tokens = re.sub(regex_p, "", content).split()
    filtered_tokens = [t for t in tokens if t not in STOPWORDS]

    return filtered_tokens


def ptokenize(html_list: list[Path], doc_idx: dict[str, list[str]]):
    for path in html_list:
```

```
        token_list = html_tokenizer(str(path))
        doc_idx[path.stem] = token_list
```

To better optimize the system during this index creation, the process of tokenizing each document into their document-token list dictionary is threaded and made adaptive based on the number of the system's CPU cores. This is done to speed up the processing time of HTML file reads and processing potentially thousands of pages.

```
class Index:
    def __init__(
        self, idx_file: Path = Path(INV_IDX_FILE), corpus_path: Path =
CRWL_OUTPUT
    ):
        self.index_path: Path = INDX_OUTPUT / idx_file
        self._magnitude_cache = {}  # inshallah this speeds up cosine
search

        if not self.index_path.exists():
            print(f"No inverted index found @ {self.index_path}, creating
one...")
            self.create_index(corpus_path)
        else:
            self.load_index()

    def create_index(self, corpus_path: Path):
        if self.index_path.exists():
            raise FileExistsError(
                f"File already exists at {str(self.index_path)}\n     Use
load_index() to load file into object instead."
            )

        self.corpus_path: Path = corpus_path
        self.corpus_mapping: Path = self.corpus_path / "url_map.jsonl"

        # run the crawler first, moron
        if not self.corpus_mapping.exists():
            raise FileNotFoundError(
                f'Expected a "url_map.jsonl" file at @
{self.corpus_mapping}, found none'
            )

        # sort so its not so random for my sake
        docs = sorted((self.corpus_path / "html").iterdir())
        self.corpus_size: int = len(docs)

        chunk_size = len(docs) // CPU_COUNT
        document_chunks = chunks(docs, chunk_size)

        threads = []
        doc_idx: dict[str, list[str]] = {}
        for chunk in document_chunks:
            t = threading.Thread(target=ptokenize, args=(chunk, doc_idx))
```

```
            threads.append(t)

        for t in threads:
            t.start()

        for t in threads:
            t.join()

        self.inverted_index: dict[str, dict[str, list[int]]] =
init_invidx_tfdf(doc_idx)
        self.bigram_index: dict[str, list[str]] = init_bigram_idx(
            list(self.inverted_index.keys())
        )
        Path(INDX_OUTPUT).mkdir(parents=True, exist_ok=True)

        with open(self.index_path, "w") as idx_file:
            idx_obj = {
                "index_path": str(self.index_path),
                "corpus_path": str(self.corpus_path),
                "corpus_mapping": str(self.corpus_mapping),
                "corpus_size": self.corpus_size,
                "inverted_index": self.inverted_index,
                "bigram_index": self.bigram_index,
            }
            json.dump(idx_obj, idx_file, indent=2)

    def load_index(self):
        if not self.index_path.exists():
            raise FileNotFoundError(
                f"File not found @ {str(self.index_path)}\n Use
create_index() to generate a new inverted index file."
            )

        with open(self.index_path, "r") as idx_file:
            idx_obj = json.load(idx_file)

        self.index_path = Path(idx_obj["index_path"])
        self.corpus_path = Path(idx_obj["corpus_path"])
        self.corpus_mapping = Path(idx_obj["corpus_mapping"])
        self.corpus_size = idx_obj["corpus_size"]
        self.inverted_index = idx_obj["inverted_index"]
        self.bigram_index = idx_obj["bigram_index"]
```

To integrate all the loosely defined functions together, the Indexer class behaves as the central pipeline for inverted index creation. If a given path to a non-existent inverted index serialized into a JSON file, the index creation pipeline is invoked. The system will look for the directory of HTML files from which the crawler downloaded its results and processes each file to construct the intermediate document-token data structure. Then, it transforms that into the complete inverted index.

```
def get_idf(self, term: str):
    if term not in self.inverted_index:
```

```
            return 0

        n = self.corpus_size
        df = len(self.inverted_index[term])

        idf = log10(n / df)
        return idf

    def get_tf(self, term: str, doc: str):
        if term not in self.inverted_index:
            return 0

        return len(self.inverted_index[term][doc])
```

From the same class, the system can retrieve the required TF-IDF values on query-time, which will be necessary on the next cosine similarity score calculation.

```
    def cosine_search(self, query_tokens: list[str], k: int = 10):
        query_vector = {}
        for term in query_tokens:
            if term in self.inverted_index:
                tf = query_tokens.count(term)
                idf = self.get_idf(term)
                query_vector[term] = tf * idf

        doc_scores: dict[str, float] = {}
        for term in query_vector:
            for doc_id in self.inverted_index[term]:
                # doc vector component
                doc_tf = self.get_tf(term, doc_id)
                doc_tfidf = doc_tf * self.get_idf(term)

                # dot product component
                doc_scores[doc_id] = (
                    doc_scores.get(doc_id, 0) + query_vector[term] * doc_tfidf
                )

        query_magnitude = sum(v**2 for v in query_vector.values()) ** 0.5

        # normalizing
        for doc_id in doc_scores:
            doc_magnitude = self._get_doc_magnitude(doc_id)
            doc_scores[doc_id] /= query_magnitude * doc_magnitude

        # best to worst
        ranked = sorted(doc_scores.items(), key=lambda x: x[1], reverse=True)
        return ranked[:k]
```

The final key implementation lies in the cosine search calculation. It expects a list of a tokenized query and represent it into its weighted TF-IDF vector form. The same process follows for all candidate documents (at least one term from query present in the document), and calculate each of its dot product with respect to the query vector. The resulting value is then normalized by the magnitude of the query and document vectors. A point of concern

that became painfully obvious when running this calculation was the execution time for summing each document magnitude necessary for the normalization. To optimize this calculation, the system employs a simple caching as follows.

```
def _get_doc_magnitude(self, doc_id: str):
    if doc_id in self._magnitude_cache:
        return self._magnitude_cache[doc_id]

    magnitude_squared = 0

    # for every term in this document
    for term in self.inverted_index:
        if doc_id in self.inverted_index[term]:
            tf = self.get_tf(term, doc_id)
            idf = self.get_idf(term)
            tfidf = tf * idf
            magnitude_squared += tfidf**2

    magnitude = magnitude_squared**0.5  # sqrt

    self._magnitude_cache[doc_id] = magnitude  # put in the cache
    return magnitude
```

Within the Index class exists a variable that stores all calculated magnitude values of a document, and the values are updated whenever the function `_get_doc_magnitude` is invoked. This will first check the cache for any existing value of that specific document before proceeding with its expensive calculation. This significantly reduces time taken for sequential queries an theoretically eliminates the document magnitude calculation bottleneck as the system continues to function.

```
def bigram(term: str):
    gram_set = set()
    togram = term + "$"
    curr = "$"

    for char in togram:
        curr += char

        if not (
            "$*" == curr or "*$" == curr
        ):  # Remove k-grams for when the `$` is right beside `*`, i.e.,
wilcard begins at the beginning or end
            gram_set.add(curr)

        curr = curr[1:]

    # Filter out wildcard bigrams
    gram_set = {item for item in gram_set if "*" not in item}

    return gram_set
```

```
def init_bigram_idx(terms: list[str]):
    kgram_idx: dict[str, list[str]] = {}

    for t in terms:
        bigrams = bigram(t)

        for b in bigrams:
            if b in kgram_idx:
                kgram_idx[b].append(t)
            else:
                kgram_idx.setdefault(b, [t])

    return kgram_idx
```

The bigram index serves as a supporting data structure for query correction and wildcard matching capabilities in the search interface. Each term in the inverted index is decomposed into character bigrams. These bigrams are then indexed to map back to their source terms, creating a reverse lookup structure. This enables the system to handle misspelled queries or partial term matches by comparing the bigrams of a query term against the bigram index to find the closest matching terms in the vocabulary. When a user submits a query with typos or uses wildcard patterns, the search component can leverage this bigram index to suggest corrections or expand wildcards into concrete terms that exist in the corpus.

Search

The search component serves as the user-facing interface that orchestrates the entire information retrieval pipeline. When a user submits a query, it undergoes several preprocessing steps before retrieval and ranking.

```
def tokenizer(query: str):
    query_lowercase = query.lower()
    regex_p = r"[^\w\s]"
    tokens = re.sub(regex_p, "", query_lowercase).split()
    filtered_tokens = [t for t in tokens if t not in STOPWORDS]

    return filtered_tokens
```

The tokenizer function applies the same text processing pipeline used during indexing: converting thequery to lowercase for case-insensitive matching, removing punctuation throughregular expression substitution, splitting on whitespace, and filtering common stopwords. This consistent preprocessing ensures that query terms are represented in the same form as indexed terms, enabling accurate matching in the inverted index.

```
def query_correction(query: list[str], bigram_idx: dict[str, list[str]]):
    corrected_query: list[str] = []
    correction_flag = False  # if correction occured, flip the flag

    for query_term in query:
        bigrams = bigram(query_term)
```

```
            # all candidate terms that match the bigrams
            candidate_terms: set[str] = set()
            for bg in bigrams:
                if bg in bigram_idx:
                    candidate_terms.update(bigram_idx[bg])

            # no candidates found, keep original term
            if not candidate_terms:
                corrected_query.append(query_term)
                continue

            # best match
            min_distance = float("inf")
            best_match = query_term

            for candidate in candidate_terms:
                lev_dist = edit_distance(candidate, query_term)

                # exact match
                if lev_dist == 0:
                    best_match = candidate
                    break

                # update best match
                if lev_dist < min_distance:
                    min_distance = lev_dist
                    best_match = candidate

            if best_match != query_term:
                correction_flag = True

            corrected_query.append(best_match)

    return corrected_query, correction_flag
```

The query correction function leverages the bigram index to handle misspelled or mistyped query terms. For each term in the tokenized query, the function generates its character bigrams and looks up candidate terms in the bigram index that share those bigrams. If no candidates are found, the original term is retained unchanged. When candidates exist, the system computes the Levenshtein edit distance between the query term and each candidate to identify the closest match. An edit distance of zero indicates an exact match, immediately selecting that term. Otherwise, the candidate with the minimum edit distance becomes the corrected term. This approach balances accuracy and efficiency by first narrowing the search space through bigram matching before applying the more expensive edit distance calculation. The function returns both the corrected query and a boolean flag indicating whether any corrections occurred, allowing the system to inform users when their query has been modified.

```
def get_url_mapping(url_map_fp: Path) -> dict[str, str]:
    if not url_map_fp.exists():
        raise FileNotFoundError(f"Expected file @ {url_map_fp}, but found
```

```
nothing")

    with open(url_map_fp, "r") as f:
        return {json.loads(line)["docID"]: json.loads(line)["url"] for
line in f}
```

The URL mapping retrieval function provides the interface between internal document identifiers and their original web addresses. It reads the JSON Lines file produced by the web crawler, parsing each line to construct a dictionary that maps document UUIDs to their source URLs. This mapping is essential for presenting search results to users, as document IDs are meaningless without their corresponding web addresses. The function includes error handling to ensure the mapping file exists before attempting to read it, preventing runtime failures when the corpus has not been properly collected.

```
def query_pipeline(query: str, index: Index, url_map: dict[str, str]):
    Path(SRCH_LOGS).mkdir(parents=True, exist_ok=True)

    q_tokenized = tokenizer(query)

    if not q_tokenized:
        raise ValueError("Processed query is empty!")

    q_corrected, flag = query_correction(q_tokenized, index.bigram_index)
    q_corrected_str = " ".join(q_corrected)

    documents = index.cosine_search(q_corrected)

    # handle non-existent and empty conditions (if it ever happens)
    file_exist = SRCH_LOGS_FP.exists() and SRCH_LOGS_FP.stat().st_size > 0

    with open(SRCH_LOGS_FP, "a", newline="") as query_log:
        writer = csv.DictWriter(
            query_log,
            fieldnames=[
                "query",
                "corrected_q",
                "is_corrected",
                "docid",
                "url",
                "score",
            ],
            delimiter=";",
        )

        if not file_exist:
            writer.writeheader()

        for doc_id, score in documents:
            writer.writerow(
                {
                    "query": query,
                    "corrected_q": q_corrected_str,
```

```
                "is_corrected": flag,
                "docid": doc_id,
                "url": url_map[doc_id],
                "score": score,
            }
        )

    return documents, q_corrected, flag
```

The query pipeline function integrates all previous components into a complete workflow. It accepts a raw user query, the index object, and the URL mapping, then orchestrates the following sequence: tokenization of the input query, correction of any misspelled terms using the bigram index, execution of the cosine similarity search to retrieve and rank relevant documents, and logging of the complete query session. The logging mechanism records the original query, corrected query, correction status, and all retrieved documents with their scores and URLs. This data is written to a CSV file with semicolon delimiters, maintaining a persistent record of system usage that can be analyzed for performance evaluation or debugging purposes. The function returns the ranked document list along with the corrected query and correction flag, providing the caller with all necessary information to present results to the user and indicate when automatic corrections have been applied. Error handling ensures that empty queries after tokenization raise an informative exception rather than silently failing during retrieval.

Operation

Search me can be locally deployed by pulling the repository at `https://github.com/yddet-www/ir-system.git` .

Setup & Usage

1. After cloning the repository, change your terminal's working directory to the root of the cloned project.

   ```
   cd /path/to/ir-system
   ```

2. With Python installed in the system, run:

   ```
   pip install -r requirements.txt
   ```

   This step will install the necessary packages used in Searchme's system. It is advised to first create a Python virtual environment beforehand.

3. Since a fresh project will not contain any document corpus, we must first execute the system's web crawler. Users are free to modify `urls.txt` file to specify the spider's seed URL. To execute the spider:

   ```
   python -m scripts.web-crawler.init
   ```

   This executes a predefined script to run the spider with its default settings.

4. Once the spider completes its execution, the system will now have its corpus of documents ready to begin its main pipeline. The search interface and inverted index creation/loading are all integrated under a single pipeline, and it initialize it:

   `python -m scripts.start-searchme`
   This script will launch the Flask-based application interface along with the core pipeline, loading the index into the system. The execution takes some time when ran for the first time as it needs to process the corpus and create an inverted index. To access the interface, open your browser of choice and visit http://127.0.0.1:5000

Test cases

This section describes the testing framework, test coverage, and evaluation methodology employed to verify Searchme's correctness and retrieval effectiveness. The testing strategy encompasses three levels: unit testing of individual components, integration testing of the complete pipeline, and information retrieval evaluation using the Cranfield benchmark dataset.

Testing Framework

Searchme's test suite is built using Python's unittest framework, organized into three primary test modules that correspond to the system's core components. All tests reside in a tests/ directory at the project root, with test fixtures stored in tests/fixtures/ to provide isolated, reproducible test environments. The test suite can be executed with standard unittest commands or with pytest for enhanced reporting capabilities. The testing infrastructure follows best practices for isolation and reproducibility. Each test class includes setup methods that initialize necessary resources such as test indices and corpus directories. Test fixtures include a small Wikipedia corpus of approximately 50 pages, a pre-built test index, and seed URLs for crawler validation. This controlled environment ensures tests run quickly and consistently across different systems without requiring large downloads or lengthy index construction.

Component Testing

Web Crawler Tests (test_crawler.py)

The web crawler test validates the complete crawling pipeline from seed URLs to downloaded HTML files. The test limits corpus collection to 50 pages to ensure reasonable test execution time. The crawler is seeded with Wikipedia URLs and configured to respect the same domain restrictions used in production.

The test verifies several critical properties of the crawler output. First, it confirms that both the HTML directory and URL mapping file are created in the expected locations. Second, it validates the integrity of the URL mapping by ensuring every downloaded HTML file has a

corresponding entry in url_map.jsonl. This bidirectional consistency check prevents orphaned files or missing mappings that would cause downstream failures in the indexer. The test leverages the get_url_mapping function from the search component, providing incidental integration testing of that utility function.

Indexer Tests (test_indexer.py)

The indexer test suite focuses on index construction correctness and persistence reliability. The primary test, test_index_identity, validates that creating a new index from a corpus produces results identical to a known-good reference index. This test first deletes any existing test index to force fresh construction, then creates a new index from the test corpus. It compares the newly created index against a pre-validated reference index stored in the fixtures directory.

The comparison operates at multiple levels of granularity. At the structural level, the test verifies that both indices contain identical sets of terms in the inverted index and identical bigrams in the bigram index. This ensures the tokenization, stopword filtering, and index construction logic produces consistent results. The test uses set comparison for efficiency, as term order is irrelevant for index correctness.

The test_idf_consistency test provides a deeper validation of index numerical properties. Rather than comparing entire data structures, it randomly samples 1000 terms from the vocabulary and verifies their IDF values match exactly between the newly created and reference indices. This statistical sampling approach balances thoroughness with execution speed, as computing IDF for every term would be expensive for large vocabularies.

The IDF consistency test serves two purposes. It validates the correctness of IDF calculation logic and confirms that index serialization and deserialization preserve numerical precision. Any discrepancies in IDF values would indicate bugs in the TF-IDF calculation, corpus size tracking, or JSON serialization process. The choice to test IDF rather than raw term frequencies reflects the importance of IDF in the ranking function—incorrect IDF values directly impact retrieval quality, while TF errors might have more localized effects.

Search Component Tests (test_search.py)

The search component test suite validates query processing, correction, and the end-to-end retrieval pipeline. The setUp method initializes a test index and URL mapping that all search tests share, improving execution efficiency by avoiding repeated index loading.

The test_tokenizer function validates query preprocessing through seven distinct test cases. These cases cover the full range of expected inputs: basic tokenization with stopword removal, case normalization, excessive whitespace handling, punctuation removal (including contractions), queries composed entirely of stopwords, and empty or whitespace-only queries. The test cases ensures the tokenizer handles edge cases gracefully, particularly the empty query scenario that could cause failures if not properly detected.

The test_query_correction function validates the spelling correction using the bigram index. Three test scenarios exercise different correction behaviors: queries with misspellings that require correction ("mattison" → "mathison"), queries with correct spellings that should pass through unchanged ("Alan Turing"), and queries with multiple errors ("hofstater" → "hofstadter"). Each test verifies both the corrected query output and the boolean correction flag, ensuring the system can accurately inform users when their queries have been modified. This transparency is important for user trust and debugging.

The test_query_pipeline function validates the complete end-to-end retrieval workflow. It submits a multi-term query and verifies the system returns exactly 10 results (the default top-K value) in descending score order. The score ordering validation is critical—incorrect ranking would render the entire system ineffective regardless of whether relevant documents are retrieved. The test includes two error condition checks: queries that tokenize to empty strings (all stopwords or whitespace) should raise ValueError exceptions with informative messages rather than silently failing or returning nonsensical results.

Information Retrieval Metric Evaluation

This subsection will walk through the evaluation procedures with live demo of core components in the system. For this purposes, the system employs Cranfield benchmark dataset.

Provided from ir-datasets.com are the following metadata information:


```
{
  "docs": {
    "count": 1400,
    "fields": {
      "doc_id": {
        "max_len": 4,
        "common_prefix": ""
      }
    }
  },
  "queries": {
    "count": 225
  },
  "qrels": {
    "count": 1837,
    "fields": {
      "relevance": {
        "counts_by_value": {
          "2": 387,
          "3": 734,
          "4": 363,
          "-1": 225,
          "1": 128
        }
```

```
      }
    }
  }
}
```

The following subsection will outline the implementation of the information retrieval metric evaluation using the Cranfield benchmark dataset, a standard test collection in information retrieval research consisting of 1,400 ddocuments, 225 queries, and relevance judgments. First, the dataset is morphed into the expected form of the system. This is achieved by loading a placeholder inverted index and replacing the data structures stored in the Index object with those formed by Cranfield's dataset, where each document's title, text, and author fields are tokenized and indexed using the same pipeline applied to the Wikipedia corpus. Then, we perform the standard query-document retrieval pipeline for all listed queries from the benchmark, retrieving the top 1000 ranked documents for each query using cosine similarity scoring. The retrieved results are compared against Cranfield's relevance judgments, which rate documents on a scale from -1 (not relevant) to 4 (complete answer), with scores 1 and above treated as relevant for evaluation purposes. The evaluation calculates standard information retrieval metrics at multiple cutoff points to assess both the system's ability to identify relevant documents and its effectiveness at ranking them highly. The three main measures we are evaluating are precision at K (the proportion of retrieved documents that are relevant), recall at K (the proportion of relevant documents that are retrieved), and mean average precision (MAP, measuring ranking quality across all queries).

In [ ]:
```
%pip install -r ../requirements.txt
```

In [1]:
```
import os
from pathlib import Path

os.chdir(Path.cwd().parent)
```

In [6]:
```
import ir_datasets
import numpy as np
import pandas as pd
from pathlib import Path
from collections import defaultdict
from src.indexer.indexme import Index, init_bigram_idx, init_invidx_tfdf
from src.search.searchme import tokenizer

# 1. LOAD AND PREPARE CRANFIELD INDEX
dataset = ir_datasets.load("cranfield")

cranfield_docidx = {
    doc.doc_id: tokenizer(f"{doc.title} {doc.text} {doc.author}")
    for doc in dataset.docs_iter()
}

cranfield_invidx = init_invidx_tfdf(cranfield_docidx)
cranfield_bigram = init_bigram_idx(list(cranfield_invidx.keys()))

dummyidx_fp = Path.cwd() / "tests" / "fixtures" / "indexer" / "dummy_index.json"
```

```
if not dummyidx_fp.exists():
    raise FileNotFoundError(
        f"Current working directory is misconfigured to {Path.cwd()}\n"
        "Please restart the notebook's kernel and start execution from the top."
    )

cranfield_idxobj = Index(dummyidx_fp)
cranfield_idxobj.inverted_index = cranfield_invidx
cranfield_idxobj.bigram_index = cranfield_bigram
cranfield_idxobj.corpus_size = len(cranfield_docidx)

print(f"Loaded Cranfield index:")
print(f"  Documents: {cranfield_idxobj.corpus_size}")
print(f"  Vocabulary size: {len(cranfield_idxobj.inverted_index)}")
print(f"  Bigrams: {len(cranfield_idxobj.bigram_index)}")
```

```
Loaded Cranfield index:
  Documents: 1400
  Vocabulary size: 10586
  Bigrams: 925
```

In [7]:
```
# 2. LOAD QUERIES AND RELEVANCE JUDGMENTS

queries = {query.query_id: query.text for query in dataset.queries_iter()}
print(f"\nLoaded {len(queries)} queries")

# consider relevance >= 1 as relevant (exclude -1)
qrels = defaultdict(set)
for qrel in dataset.qrels_iter():
    if qrel.relevance >= 1:
        qrels[qrel.query_id].add(qrel.doc_id)

print(f"Loaded relevance judgments for {len(qrels)} queries")
```

```
Loaded 225 queries
Loaded relevance judgments for 225 queries
```

In [8]:
```
# 3. RUN RETRIEVAL FOR ALL QUERIES

def retrieve_documents(query_text, index_obj, k=1000):
    tokens = tokenizer(query_text)
    if not tokens:
        return []
    results = index_obj.cosine_search(tokens, k=k)
    return results  # list of (doc_id, score) tuples


all_results = {}
for query_id, query_text in queries.items():
    results = retrieve_documents(query_text, cranfield_idxobj, k=1000)
    all_results[query_id] = results

print(f"\nRetrieved results for {len(all_results)} queries")
```

```
Retrieved results for 225 queries
```

In [ ]:
```
# 4. EVALUATION METRICS FUNCTIONS
```

```python
def get_precision(results, qrels, k=10):
    precisions = []

    for query_id, doc_list in results.items():
        if query_id not in qrels:
            continue

        relevant_docs = qrels[query_id]
        top_k_docs = [doc_id for doc_id, _ in doc_list[:k]]

        relevant_retrieved = len(set(top_k_docs) & relevant_docs)
        precision = relevant_retrieved / k if k > 0 else 0
        precisions.append(precision)

    return np.mean(precisions) if precisions else 0.0


def get_recall(results, qrels, k=10):
    recalls = []

    for query_id, doc_list in results.items():
        if query_id not in qrels:
            continue

        relevant_docs = qrels[query_id]
        if len(relevant_docs) == 0:
            continue

        top_k_docs = [doc_id for doc_id, _ in doc_list[:k]]
        relevant_retrieved = len(set(top_k_docs) & relevant_docs)

        recall = relevant_retrieved / len(relevant_docs)
        recalls.append(recall)

    return np.mean(recalls) if recalls else 0.0


def get_avg_precision(retrieved_docs, relevant_docs):
    if len(relevant_docs) == 0:
        return 0.0

    relevant_retrieved = 0
    sum_precisions = 0.0

    for i, doc_id in enumerate(retrieved_docs, start=1):
        if doc_id in relevant_docs:
            relevant_retrieved += 1
            precision_at_i = relevant_retrieved / i
            sum_precisions += precision_at_i

    return sum_precisions / len(relevant_docs)


def get_map(results, qrels):
    average_precisions = []
```

```
    for query_id, doc_list in results.items():
        if query_id not in qrels:
            continue

        relevant_docs = qrels[query_id]
        retrieved_docs = [doc_id for doc_id, _ in doc_list]

        ap = get_avg_precision(retrieved_docs, relevant_docs)
        average_precisions.append(ap)

    return np.mean(average_precisions) if average_precisions else 0.0


def get_f1(results, qrels, k=10):
    f1_scores = []

    for query_id, doc_list in results.items():
        if query_id not in qrels:
            continue

        relevant_docs = qrels[query_id]
        if len(relevant_docs) == 0:
            continue

        top_k_docs = [doc_id for doc_id, _ in doc_list[:k]]
        relevant_retrieved = len(set(top_k_docs) & relevant_docs)

        precision = relevant_retrieved / k if k > 0 else 0
        recall = relevant_retrieved / len(relevant_docs)

        if precision + recall > 0:
            f1 = 2 * (precision * recall) / (precision + recall)
        else:
            f1 = 0.0

        f1_scores.append(f1)

    return np.mean(f1_scores) if f1_scores else 0.0
```

In [ ]:
```
# 5. CALCULATE ALL METRICS

print("CRANFIELD EVALUATION RESULTS")

map_score = get_map(all_results, qrels)
print(f"\nMean Average Precision (MAP): {map_score:.4f}")

print("\nPrecision@K:")
for k in [5, 10, 20, 50, 100]:
    p_at_k = get_precision(all_results, qrels, k=k)
    print(f"  P@{k:3d}: {p_at_k:.4f}")

print("\nRecall@K:")
for k in [5, 10, 20, 50, 100]:
    r_at_k = get_recall(all_results, qrels, k=k)
    print(f"  R@{k:3d}: {r_at_k:.4f}")
```

```
print("\nF1@K:")
for k in [5, 10, 20]:
    f1_at_k = get_f1(all_results, qrels, k=k)
    print(f"  F1@{k:2d}: {f1_at_k:.4f}")
```

```
CRANFIELD EVALUATION RESULTS

Mean Average Precision (MAP): 0.2656

Precision@K:
  P@  5: 0.2880
  P@ 10: 0.2169
  P@ 20: 0.1460
  P@ 50: 0.0790
  P@100: 0.0472

Recall@K:
  R@  5: 0.2408
  R@ 10: 0.3532
  R@ 20: 0.4704
  R@ 50: 0.5993
  R@100: 0.6981

F1@K:
  F1@ 5: 0.2354
  F1@10: 0.2440
  F1@20: 0.2063
```

Conclusion

This project successfully implemented a functional information retrieval system that demonstrates the fundamental principles of web search technology. Searchme integrates document collection, indexing, and query processing into a cohesive pipeline capable of retrieving relevant documents from a corpus of web pages. The system employs breadth-first web crawling for corpus collection, inverted indexing with positional information for efficient retrieval, TF-IDF weighting for term importance, and cosine similarity for relevance scoring. Practical optimizations including bigram-based spelling correction and magnitude caching enhance both user experience and system performance.

Evaluation Summary

The Cranfield benchmark evaluation produced the following results across 225 queries and 1,400 documents:

Ranking Quality:

Mean Average Precision (MAP): 0.27 Precision@5: 0.29, Precision@10: 0.22, Precision@20: 0.15

Retrieval Coverage:

Recall@5: 0.24, Recall@10: 0.35, Recall@20: 0.47, Recall@100: 0.70

Combined Metrics:

F1@5: 0.24, F1@10: 0.24, F1@20: 0.21

These results place Searchme above random retrieval and simple Boolean systems, validating that the TF-IDF vector space model produces meaningful relevance rankings. The MAP of 0.27 exceeds typical baselines (0.15-0.20) but falls short of sophisticated ranking functions like BM25 (0.40-0.45). The relatively high recall at 100 documents (70%) demonstrates the system successfully identifies most relevant documents in the corpus, while the low precision at top ranks (22% at position 10) reveals significant weaknesses in ranking quality.

Limitations and Future Work

The system lacks several capabilities that would improve effectiveness: stemming for handling morphological variants, query expansion for addressing vocabulary mismatch, phrase matching for multi-word concepts, and field-weighted indexing for document structure awareness. The ranking model considers only term statistics without document quality signals or user interaction data. The JSON-based index persistence limits scalability to very large corpora, and the static corpus assumption prevents incremental updates.

Future enhancements should prioritize implementing stemming, replacing TF-IDF with BM25, and adding query expansion. Leveraging the existing positional information for phrase queries and proximity scoring would improve precision for multi-word concepts. Performance optimizations including index compression and parallel query processing would support higher throughput and larger corpora.

Closing Remarks

Searchme successfully demonstrates classical information retrieval techniques for web document search, achieving functional correctness and baseline effectiveness. The measured performance aligns with expectations for a TF-IDF system, validating both implementation and evaluation methodology. While falling short of modern search engines, the system establishes a functional foundation with well-documented limitations and clear improvement pathways. The modular architecture, comprehensive testing, and reproducible evaluation provide a solid platform for future enhancements and serve as an educational example of core IR principles in practice.

Source code

Repository for this sytem can be found at https://github.com/yddet-www/ir-system

Cranfield dataset is generously provided from https://ir-datasets.com/

Bibliography

1. W. B. Croft, D. Metzler, and T. Strohman, Search Engines: Information Retrieval in Practice. Boston, MA: Addison-Wesley, 2009.

2. C. D. Manning, P. Raghavan, and H. Schütze, An Introduction to Information Retrieval. Cambridge, England: Cambridge University Press, 2008.

3. S. MacAvaney, A. Yates, S. Feldman, D. Downey, A. Cohan, and N. Goharian, "Simplified data wrangling with ir_datasets," in Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval, 2021, pp. 2429-2436.