Information Retrieval Project Report

By Elijah Hoedl

Abstract

This project is an end-to-end Information Retrieval system consisting of a web crawler, TF-IDF indexer, query processor with spelling correction, and a Flask-based search API. The system crawls a website, extracts and downloads HTML documents, builds searchable vector-space representations, and returns ranked retrieval results based off of cosine similarity.

The goal of the project is to build a workable search engine that can process a corpus of documents and return the most relevant documents from a particular query.

There are a few areas where further developments would improve the project. First would be to implement a Scrapy-based crawler instead of the custom crawler currently in use, as I could not get the Scrapy-based crawler to run properly. The crawler currently in use functions well with a small corpus of documents, but would end up taking a long time to crawl a large corpus of documents as it processes them one at a time, not concurrently as Scrapy can d.

Another way to develop the project further would be to include query expansion using WordNet. This would make it so that words similar to the one(s) in the query will also be included in the processing and could also factor into what files are returned at the end. This extension allows for more relevant results to be returned even if the precise word does not line up. For example, if the query is [“history”], then results could be returned that include words like [“history”, “historical”, “account”, “record”, “chronicle”, “past”] and so on. The query expansion gives users more allowance in what exactly they are searching for and returns a more robust and relevant set of results.

Overview

This project consists of crawling websites and downloading respective html files. It then goes on to process and index data, creating a TF-IDF matrix and saving said index as a json file. The system then uses a query processor to both rank relevant documents and save a csv file of top k results. The query preoccesor also includes a spell checker that uses minimum edit distance (Levenshtein distance) to process the closest word to the one being queried when said word does not appear in the corpus. This approach is based on material from *Introduction to Information Retrieval* by Christopher D. Manning, Prabhakar Raghavan and Hinrich Schütze.

Design

The system has the following key capabilities:

- Domain-limited crawling with manipulatable depths and page limits
- html processing to extract and normalize text
- TF-IDF index construction for a vector-space retrieval
- Query processing with a spelling correction
- Ranked relevant results for query
- Flask API that uses json-based searches

The different components of the system interacts as follows:

- The crawler downloads and stores html documents
- The indexer process the corpus of html documents and builds vector-based representations
- The query processor turns user queries into TF-IDF vectors
- Results are returned either through a Flask API or through a command line interface

Architecture

The project consists of three important classes: Crawler, Indexer, and QueryProcessor. The Crawler class extracts data by crawling websites and stores each site as an html file. The Indexer class parses through each stored html and creates a TF-IDF matrix and is saved as an index in a json file. The QueryProcessor class then loads the index, preprocesses provided queries, spell checks invalid inputs, and performs ranking using TF-IDF and cosine similarity scores to return relevant documents. The Flask application is also used to search using an interactable command line interface.

The system utilizes the following Python packages: os, json, csv, re, numpy, sklearn, bs4, flask, requests, urllib, nltk, shutil.

Operation

To begin the process you must install the following packages: import os
import json
import csv
import re
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
from bs4 import BeautifulSoup
from flask import Flask, request, jsonify
import requests

```
from urllib.parse import urljoin, urlparse
from nltk.metrics import edit_distance
import shutil
```

Then, you enter in the site you are looking to scrape in this line, using https://books.toscrape.com/ as an example.

```
site = "https://books.toscrape.com/"
```

After that, you set the crawler to your desired maximum number of pages and depth and run the crawl command:

```
crawler = Crawler(site, max_pages=10, max_depth=2)
crawler.crawl()
```

Then, you initialize the indexer object and run the command to build the index:

```
indexer = Indexer()
indexer.build_index()
```

After this, you set the query to what you are looking to search for – in this instance in incorrectly spelled word “histrorical” to test the spelling correction – and run the command that will save those results to a csv file:

```
query = ["Histrorical"]
save_query_results_csv(query)
```

After this, you can then run the following code to begin the Flask application which will allow you to search for results directly to and from the terminal:

```
app = Flask(__name__)
query_processor = QueryProcessor(index_file='index.json')

@app.route('/search', methods=['POST'])
def search_api():
    data = request.get_json()
    query = data.get("query", "")
    top_k = data.get("top_k", 5)
    results = query_processor.search(query, top_k=top_k)
    return jsonify(results)
```

```
if __name__ == "__main__":
    app.run(port=5000, debug=False)

stop = False
while not stop:
    print("Enter Query:")
    query = input()
    print(f"Searching for {query}")
    response = requests.post(
        "http://localhost:5000/search",
        json={"query": query, "top_k": 3}
    )
    print(response.json())
    print("Continue? (Y/N)")
    cont = input()
    while not stop:
        if cont == "Y":
            stop = False
            break
        if cont == "N":
            stop = True
            print("Thanks for searching!")
        else:
            print("Invalid input.")
            print("Continue? (Y/N)")
            cont = input()
```

From there, you can continue searching queries until you choose to stop and end the program.

Conclusion

Overall, the project was successful in modeling a small-scale, end-to-end search engine. It crawls websites, downloads them as html files, creates a TF-IDF matrix and saves it to an index as a json file, and performs ranked retrieval using TF-IDF values and cosine similarity.

Data Sources

Link to root website for crawling: https://books.toscrape.com/
Books to Scrape is a fictitious website designed for the purpose of letting people practice web scraping in a safe and legal way.

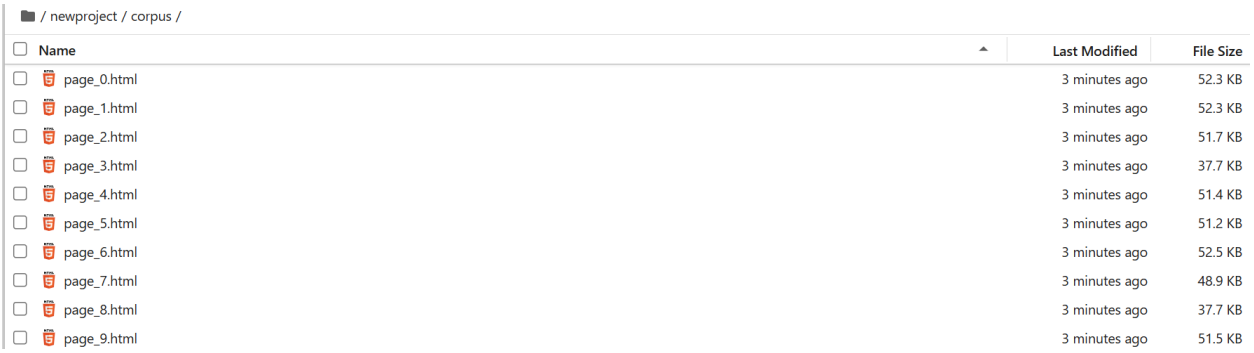
Further data is acquired through the crawling process.

Test Cases

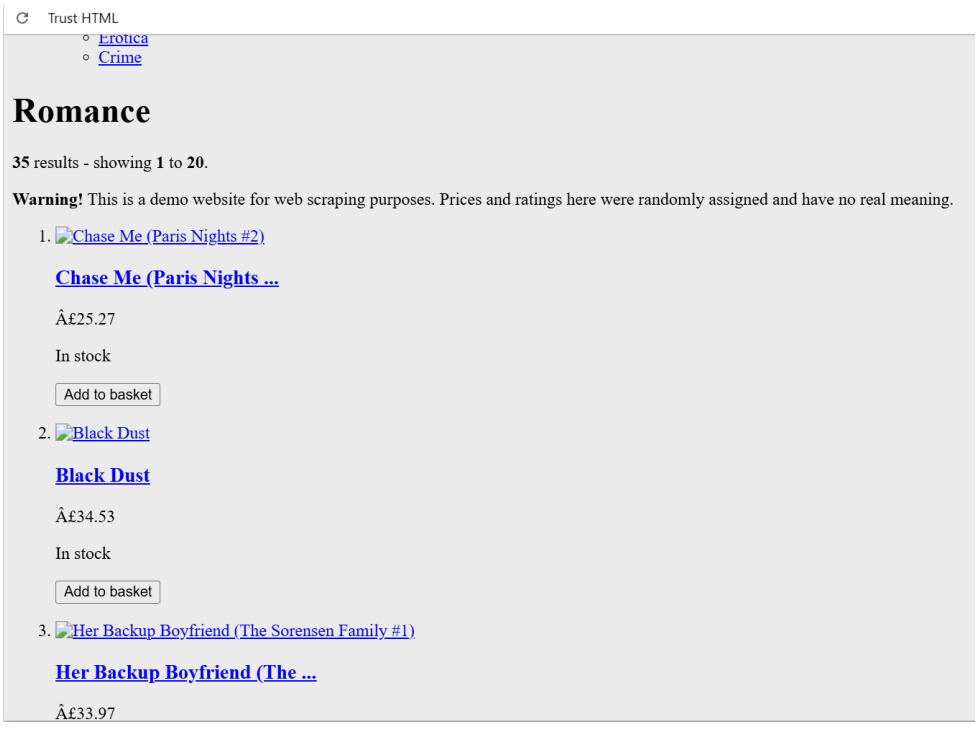
Testing the crawler using these parameters:

site = "https://books.toscrape.com/"
crawler = Crawler(site, max_pages=10, max_depth=2)
crawler.crawl()

We then go to newly made folder called “corpus” and find the following:



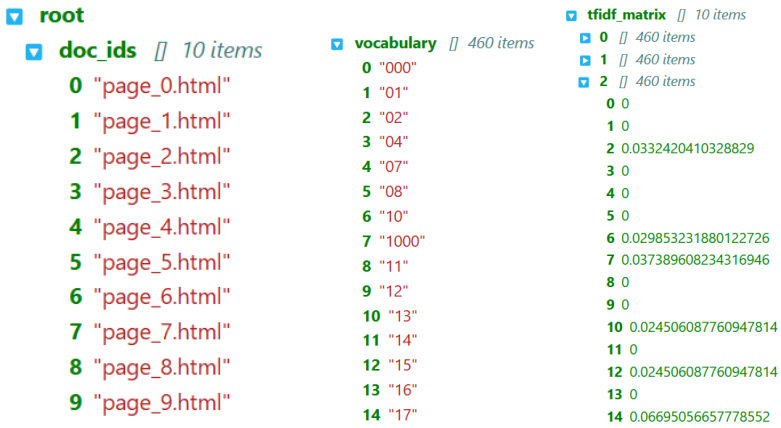
We can then open up page_9.html as an example of what was downloaded:



Since the crawler ran successfully, we can now build our index:

indexer = Indexer()
indexer.build_index()

This creates the following json file which contains the document ids, the vocabulary of the corpus, and the TF-IDF matrix:



Now we save our results to a csv file and get the following:

	query	doc_id	score
1	Histrorical	page_5.html	0.08752523499162378
2	Histrorical	page_8.html	0.05848660342984474
3	Histrorical	page_3.html	0.05601379570814349
4	Histrorical	page_7.html	0.03799581226872093
5	Histrorical	page_2.html	0.03717398219224219

Although the word “Histrorical” does not appear in the vocabulary, the next closest word “Historical” does and that is what is being used to search. We see that in what is returned when the code is run:

```
# Test Code
site = "https://books.toscrape.com/"
crawler = Crawler(site, max_pages=10, max_depth=2)
crawler.crawl()

indexer = Indexer()
indexer.build_index()

query = ["Histrorical"]
save_query_results_csv(query)
```

```
Crawling 1/10
Crawling 2/10
Crawling 3/10
Crawling 4/10
Crawling 5/10
Crawling 6/10
Crawling 7/10
Crawling 8/10
Crawling 9/10
Crawling 10/10
Index saved to index.json
Did you mean: historical? Using corrected term.
Query results saved to query_results.csv
```

The following showcases the command-line inputs for searching using the Flask application. It shows what the results are for a word that is incorrectly spelled and multiple words in one query as well as those same words split apart into two separate queries to ensure that the results are truly for the two words combined:

```
Enter Query:
 Mystery Books
Searching for Mystery Books
127.0.0.1 - - [07/Dec/2025 12:27:16] "POST /search HTTP/1.1" 200 -
[{'doc_id': 'page_8.html', 'score': 0.103390684734529}, {'doc_id': 'page_4.html', 'score': 0.10027364908523356}, {'doc_id': 'page_3.html', 'score': 0.0
9901933696306549}]
Continue? (Y/N)
 Y
Enter Query:
 Mystory Books
Searching for Mystory Books
127.0.0.1 - - [07/Dec/2025 12:27:35] "POST /search HTTP/1.1" 200 -
Did you mean: mystery? Using corrected term.
[{'doc_id': 'page_8.html', 'score': 0.103390684734529}, {'doc_id': 'page_4.html', 'score': 0.10027364908523356}, {'doc_id': 'page_3.html', 'score': 0.0
9901933696306549}]
Continue? (Y/N)
 Y
Enter Query:
 Black Dust
Searching for Black Dust
127.0.0.1 - - [07/Dec/2025 12:29:00] "POST /search HTTP/1.1" 200 -
[{'doc_id': 'page_9.html', 'score': 0.07800889886156678}, {'doc_id': 'page_2.html', 'score': 0.023505672634832983}, {'doc_id': 'page_1.html', 'score':
0.023233711993627167}]
Continue? (Y/N)
 Y
Enter Query:
 Black
Searching for Black
127.0.0.1 - - [07/Dec/2025 12:29:07] "POST /search HTTP/1.1" 200 -
[{'doc_id': 'page_9.html', 'score': 0.06281930262283093}, {'doc_id': 'page_2.html', 'score': 0.03324204103288293}, {'doc_id': 'page_1.html', 'score':
0.03285743060565798}]
Continue? (Y/N)
 Y
Enter Query:
 Dust
Searching for Dust
127.0.0.1 - - [07/Dec/2025 12:29:13] "POST /search HTTP/1.1" 200 -
[{'doc_id': 'page_9.html', 'score': 0.047501940132987905}, {'doc_id': 'page_8.html', 'score': 0.0}, {'doc_id': 'page_7.html', 'score': 0.0}]
Continue? (Y/N)
 INCORRECTINPUT
Invalid input.
Continue? (Y/N)
 N
Thanks for searching!
```

Source Code:

The code for this project was in part generated using the assistance of Microsoft Copilot.

The project requires the following open-source dependencies: Python 3.12+, scikit-learn 1.6+, Flask 3.1+, BeautifulSoup4, Requests, NLTK

Bibliography:

Manning, Christopher D., Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.