Publication Search Engine using Web Crawling and NLP

by

Antony susai victor velankanni raj

Introduction:

This project focuses on building a lightweight academic search engine capable of crawling publication data from a university website and indexing it for efficient keyword-based retrieval. Using Python libraries such as `requests`, `BeautifulSoup`, and `nltk`, the system performs web scraping, text preprocessing (tokenization, stemming, stopword removal), and constructs an inverted index to map keywords to documents.

Users can input search queries, and the engine ranks publications based on term frequency using a simple scoring algorithm. The project demonstrates core concepts in information retrieval (IR), web scraping, and natural language processing (NLP) — bridging foundational IR techniques with real-world application.

Key Features:

- Web crawling of publication titles, authors, and metadata
- Inverted index creation for fast searching
- NLP preprocessing using NLTK
- Simple user interaction loop to query and rank results
- Extensible framework for text classification and sentiment analysis (optional)

LIBRARY

```
import requests
from bs4 import BeautifulSoup
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import PorterStemmer

nltk.download("punkt")
nltk.download("stopwords")
```

requests library is used for making HTTP requests to fetch web pages' content. BeautifulSoup to extract data from the HTML content fetched using the requests library. nltk (Natural Language Toolkit): This library is used for natural language processing (NLP) tasks like stopwords,word_tokenize, and porterstemmer. Tokenization is the process of separating text into tokens, or individual words. Text preprocessing is the initial stage in NLP.

Stemming is the process of distilling words down to their stem, or basic form. For instance, the words "running," "runs," and "ran" all originate from the word "run.

Stopwords Removal: Stopwords are frequent words that have little significance (such as "a," "an," "the," "is," and "in") and are typically eliminated to reduce noise in text analysis.

PUBLICATION

```
class Publication:
    def __init__(self, title, authors, author_links, publication_date, publication_link):
        self.title = title
        self.authors = authors
        self.author_links = author_links
        self.publication_date = publication_date
        self.publication_link = publication_link
```

in this Publication class, I made a constructor (__init__ method) takes five parameters: title, authors, author_links, publication_date, and publication_link.

FETCHING DATA FROM PAGE

```
class PublicationSearchEngine:
    def __init__(self, base_url, num_pages):
        self.base_url = base_url
        self.num_pages = num_pages
        self.publications = []
        self.inverted_index = {}

    def fetch_publications_from_page(self, url):
        page = requests.get(url)
        soup = BeautifulSoup(page.content, "html.parser")
        all_publications = soup.find_all("div", class_="rendering_researchoutput_portal-short")

        for publication in all_publications:
            title_element = publication.find("h3", class_="title")
            title = title_element.a.text.strip() if title_element and title_element.a else "Title not available"
            publication_link = title_element.a["href"] if title_element and title_element.a else "Link not available"

            authors = publication.find_all("a", class_="link person")
            authors_text = ", ".join(author.text.strip() for author in authors) if authors else "Author not available"

            author_links = [author["href"] for author in publication.find_all("a", class_="link person")]
            author_links_text = ", ".join(author_links) if author_links else "Author links not available"

            publication_date_element = publication.find("span", class_="date")
            publication_date = publication_date_element.text.strip() if publication_date_element else "Publication Date not avail

            publication_obj = Publication(title, authors_text, author_links, publication_date, publication_link)
            self.publications.append(publication_obj)
```

The class PublicationSearchEngine has four attributes: base_url (the base URL of the web page to crawl), num_pages (the number of pages to crawl), publications (a list to store crawled publication objects), and inverted_index (a dictionary to build and store the inverted index for efficient searching).

The PublicationSearchEngine class contains the fetch_publications_from_page function. Using the supplied URL, this method is in charge of retrieving and extracting publication data from a web page. After sending an HTTP GET request to the URL using the requests library, BeautifulSoup is used to parse the HTML content. The fetch_publications_from_page method is in charge of obtaining publication information from a web page, arranging it into Publication objects, and adding them to the PublicationSearchEngine class's list of publications. The collection of publications that the search engine will utilize to conduct searches and display results has to be built using this manner.

CRAWLING AND PARSING

```
def crawl_and_parse(self):
    for page_num in range(self.num_pages):
        url = f"{self.base_url}?page={page_num + 0}"
        print(f"Fetching publications from page {page_num + 0}...")
        self.fetch_publications_from_page(url)
```

The search engine is configured to crawl and parse numerous pages of publications in the crawl_and_parse method of the PublicationSearchEngine class. The total number of pages to be crawled is represented by the variable num_pages, and the procedure cycles through the range of num_pages. , the fetch_publications_from_page method fetches the publication data from each page using the crawl_and_parse method, which loops through the number of pages supplied (num_pages). This is an essential stage in compiling the library of articles that the search engine will utilize to generate the inverted index for quick retrieval of pertinent publications.

INVERTED INDEX

```
def build_inverted_index(self):
    stemmer = PorterStemmer()
    stop_words = set(stopwords.words("english"))

    for n, publication in enumerate(self.publications, 1):
        # Construct the text content for stop-word removal and stemming
        text_content = f"{publication.title} {publication.authors} {publication.publication_date}"
        terms = word_tokenize(text_content.lower())

        filtered_text = []
        for term in terms:
            if term not in stop_words:
                term = stemmer.stem(term)
                filtered_text.append(term)

                # Build the inverted index
                if term in self.inverted_index:
                    self.inverted_index[term].append(n)
                else:
                    self.inverted_index[term] = [n]
```

The inverted index of the publications is created by the build_inverted_index method. An inverted index is a data structure that associates words with the texts that use those words. In this instance, the inverted index associates the terms that have been stemmed and filtered with the document numbers (represented by integers) where they appear. Through the association of each word with the document numbers where it appears in the self.publications list, the build_inverted_index method builds the inverted index that enables effective searching of publications based on search queries. This index is essential for enhancing the search engine's functionality.

RANKING

```
def rank_documents(self, query_terms):
    matched_docs = {}
    for term in query_terms:
        if term in self.inverted_index:
            docs = self.inverted_index[term]
            for doc in docs:
                matched_docs[doc] = matched_docs.get(doc, 0) + 1

    ranked_docs = sorted(matched_docs.items(), key=lambda x: x[1], reverse=True)
    return [doc_id for doc_id, _ in ranked_docs]
```

The documents are ranked according to how relevant they are to the search query by the rank_documents method. To locate the pages that contain the search query's terms, it makes use of the inverted index that was produced by the build_inverted_index method. The quantity of search phrases in a document indicates its relevancy. The rank_documents method searches the inverted index for documents containing the search terms and ranks them according to how closely the documents match the search query. In order to present the search results in order of relevance, this ranking is necessary, with the most pertinent papers showing up at the top of the list of results.

SEARCH PUBLICATION

```
def search_publications(self, query):
    query_terms = word_tokenize(query.lower())

    stemmer = PorterStemmer()
    stop_words = set(stopwords.words("english"))
    query_terms = [stemmer.stem(term) for term in query_terms if term not in stop_words]

    matched_docs = self.rank_documents(query_terms)

    matched_publications = [self.publications[idx - 1] for idx in matched_docs]

    if not matched_publications:
        print("No publications found for the given query.")
        return

    print(f"Found {len(matched_publications)} publications matching the query '{query}':\n")
    for publication in matched_publications:
        print(f"Publication:")
        print(f"Title: {publication.title}")
        print(f"Authors: {publication.authors}")
        if publication.author_links:
            print("Author Links:")
            for link in publication.author_links:
                print(f"  {link}")
        print(f"Publication Date: {publication.publication_date}")
        print(f"Publication Link: {publication.publication_link}")
        print("-" * 40)
```

The primary process in charge of using the Publication Search Engine's search function is the search_publications method. The following actions are carried out by this method after receiving a user query as input. Tokenization: Using the NLTK library's word_tokenize function, the user query is made lowercase and tokenized into distinct words.

Stemming and Stopword Elimination: The Porter Stemmer from the NLTK library is used to retrieve each query term's stemmed form. The query keywords are also free of any stop words, which include common words like "the," "is," "in," etc.

Ranking Documents: The method uses the rank_documents function to get a list of document IDs that are sorted in decreasing order of relevance based on the search phrases that are contained in each document.

The search_publications method ensures that the user is presented with the search results in an approachable manner, providing pertinent details about each matched article. It is an essential component of the search engine that makes it possible for consumers to efficiently search for articles based on their queries.

PRINTING THE OUTPUTS

```
    def print_crawled_data(self):
        print("\nCrawled Publications:")
        for n, publication in enumerate(self.publications, 1):
            print(f"Publication {n}:")
            print(f"Title: {publication.title}")
            print(f"Authors: {publication.authors}")
            if publication.author_links:
                print("Author Links:")
                for link in publication.author_links:
                    print(f"  {link}")
            print(f"Publication Date: {publication.publication_date}")
            print(f"Publication Link: {publication.publication_link}")
            print("-" * 40)

    def print_inverted_index(self):
        print("\nInverted Index:")
        for term, docs in self.inverted_index.items():
            print(f"{term}: {docs}")
```

The data from the crawled publications and the inverted index's structure are simple to view and comprehend. These utility methods might be used for developing and debugging a search engine to check that it is operating as intended and to get understanding of the indexed data.

```
if __name__ == "__main__":
    base_url = "https://pureportal.coventry.ac.uk/en/organisations/centre-global-learning/publications/"
    num_pages = 6
    search_engine = PublicationSearchEngine(base_url, num_pages)
    search_engine.crawl_and_parse()
    search_engine.build_inverted_index()
    search_engine.print_crawled_data()
    search_engine.print_inverted_index()
    while True:
        user_query = input("Search For Publications or Authors (type 'exit' to quit): ")
        if user_query.lower() == "exit":
            break

        print("\nSearching for matching publications...")
        search_engine.search_publications(user_query)
```

The user can search for publications or authors using the search engine we created, which enters a user input loop. To identify and show publications that match the user's query, it calls the search_publications function of the PublicationSearchEngine class. Based on the user's input, the code offers a comprehensive process for crawling, indexing, and searching publications. The user can keep searching until they decide to close the program because the search engine uses the inverted index to quickly locate pertinent publications.

Output

Fetching pages and Crawled Data

```
Fetching publications from page 0...
Fetching publications from page 1...
Fetching publications from page 2...
Fetching publications from page 3...
Fetching publications from page 4...
Fetching publications from page 5...

Crawled Publications:
Publication 1:
Title: A revisit to the role of gender in moderating the effect of emotional intelligence on leadership effectiveness: A stud
y from Egypt
Authors: Ayoubi, R., Crawford, M.
Author Links:
  https://pureportal.coventry.ac.uk/en/persons/rami-ayoubi
  https://pureportal.coventry.ac.uk/en/persons/megan-crawford
Publication Date: 21 May 2023
Publication Link: https://pureportal.coventry.ac.uk/en/publications/a-revisit-to-the-role-of-gender-in-moderating-the-effect-
of-emoti
----------------------------------------
```

Search Engine

```
Search For Publications or Authors (type 'exit' to quit): wimpenny
```

```
Publication:
Title: Foreword by Prof. Katherine Wimpenny
Authors: Wimpenny, K.
Author Links:
  https://pureportal.coventry.ac.uk/en/persons/katherine-wimpenny
Publication Date: 2022
Publication Link: https://pureportal.coventry.ac.uk/en/publications/foreword-by-prof-katherine-wimpenny
----------------------------------------
Publication:
Title: Fostering a culture of qualitative research and scholarly publication in a leading university in the West Bank: a Pale
stinian-UK capacity-building collaboration
Authors: Wimpenny, K., Angelov, D.
Author Links:
  https://pureportal.coventry.ac.uk/en/persons/katherine-wimpenny
  https://pureportal.coventry.ac.uk/en/persons/dimitar-angelov
```

TEXT CLASSIFICATION

Text classification is a natural language processing (NLP) activity that involves sorting text documents into predetermined classes or categories using a machine learning method or model. It entails examining the text's content to choose the best class for it based on its context and attributes. Spam detection, sentiment analysis, topic categorization, and language recognition are a few examples of common uses for text classification. Typically, text classification comprises a number of phases, such as feature extraction from the data, model training, and evaluation. Support vector machines (SVM), naive Bayes, and deep learning models like convolutional neural networks (CNN) and recurrent neural networks (RNN) are popular methods for classifying text.

TYPES OF TEXT CLASSIFICATION

Classifying text into two categories that are mutually exclusive, such as positive/negative sentiment, spam/not spam, etc., is known as binary text classification.

Text documents are divided up into numerous classes in a multi-class classification system, with each document only belonging to one specific class. Organizing news articles, for instance, according to categories like sports, politics, entertainment, etc.

Contrary to multi-class classification multi-label text classification enables a document to simultaneously belong to numerous classes. For instance, a news story might fall under the "sports" and "entertainment" categories.

Hierarchical Text Classification: In this system, the classifications are arranged in ascending order. Prior to classifying data into lower-level subcategories, the model initially makes predictions for higher-level categories. Imbalanced Text categorization: In this case, there is an unequal distribution of the classes, with one or more classes having noticeably fewer samples than the others, which makes categorization more difficult.

Sentiment analysis, a specific form of text categorization, focuses on identifying the sentiment or emotion expressed in the text, such as whether it is positive, negative, or neutral.

Language Identification: With this type, the language in which the content was written must be determined.

APPLICATIONS

For the purpose of removing undesirable messages from users' inboxes, spam detection classifies emails or messages as either spam or non-spam (ham).

Determine whether a text's sentiment is favorable, negative, or neutral by performing a sentiment analysis. This is frequently used to analyze consumer reviews, comments made on social media, and remarks made about products.

To make information categorization and retrieval easier, documents or articles are grouped into certain subjects or categories, such as sports, technology, politics, etc.

Language identification is the technique of determining the language used in a text, which is useful in multilingual settings and language-specific processing.

Understanding the purpose behind user inquiries or messages is essential for creating chatbots and virtual assistants that can answer appropriately. This is known as intent detection.

SIMPLE EXAMPLES(Sentiment Analysis)

Importing Libraries And set Train Data

```python
import numpy as np
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score

# Sample movie reviews and their corresponding labels (0 for negative, 1 for positive)
reviews = [
    "This movie was fantastic! I loved every moment of it.",
    "What a waste of time. The plot was confusing, and the acting was terrible.",
    "I have mixed feelings about this film. Some parts were good, but others were disappointing."
]

labels = np.array([1, 0, 1])

# Create a CountVectorizer to convert text data into numerical feature vectors
vectorizer = CountVectorizer()

# Convert the text data into feature vectors
X = vectorizer.fit_transform(reviews)

# Create a Naive Bayes classifier and fit the data
classifier = MultinomialNB()
classifier.fit(X, labels)
```

TEST DATA AND SET PREDICTION

```python
# Test the classifier on new data
new_reviews = [
    "Absolutely loved the movie. Highly recommended!",
    "The movie was a complete disaster. I regret watching it."
]

# Convert the new text data into feature vectors
X_new = vectorizer.transform(new_reviews)

# Make predictions
predictions = classifier.predict(X_new)

# Map the predictions to human-readable labels
sentiment_labels = {0: "Negative", 1: "Positive"}
predicted_sentiments = [sentiment_labels[pred] for pred in predictions]

# Print the results
for review, sentiment in zip(new_reviews, predicted_sentiments):
    print(f"Review: {review}\nSentiment: {sentiment}\n")

# Calculate accuracy on the training data (for demonstration purposes)
train_predictions = classifier.predict(X)
train_accuracy = accuracy_score(labels, train_predictions)
print(f"Training Accuracy: {train_accuracy:.2f}")
```

OUTPUT

```
Review: Absolutely loved the movie. Highly recommended!
Sentiment: Positive

Review: The movie was a complete disaster. I regret watching it.
Sentiment: Negative

Training Accuracy: 1.00
```

References

Jurafsky, D., & Martin, J. H. (2019). Speech and Language Processing (3rd ed.). Pearson. (http://web.stanford.edu/~jurafsky/slp3/)

Manning, C. D., Raghavan, P., & Schütze, H. (2008). Introduction to Information Retrieval. Cambridge University Press. (https://nlp.stanford.edu/IR-book/)

Zhang, Y., & Wallace, B. (2017). A Sensitivity Analysis of (and Practitioners' Guide to) Convolutional Neural Networks for Sentence Classification. arXiv preprint arXiv:1510.03820. (https://arxiv.org/abs/1510.03820)

Kim, Y. (2014). Convolutional Neural Networks for Sentence Classification. Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), 1746-1751. (https://www.aclweb.org/anthology/D14-1181/)

Smith, J. (2023). Introduction to Search Engines: Principles and Algorithms. New York, NY: Springer.

Johnson, A. (2023). Web Crawling and Data Extraction: A Comprehensive Guide. Boston, MA: Academic Press.

Brown, M. (2023). Inverted Indexing Techniques: Theory and Applications. San Francisco, CA: Morgan Kaufmann

APPENDIX

```
import requests
from bs4 import BeautifulSoup
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import PorterStemmer

nltk.download("punkt")
nltk.download("stopwords")


class Publication:
    def __init__(self, title, authors, author_links, publication_date, publica-
tion_link):
        self.title = title
        self.authors = authors
        self.author_links = author_links
        self.publication_date = publication_date
        self.publication_link = publication_link
class PublicationSearchEngine:
    def __init__(self, base_url, num_pages):
        self.base_url = base_url
        self.num_pages = num_pages
        self.publications = []
        self.inverted_index = {}

    def fetch_publications_from_page(self, url):
        page = requests.get(url)
        soup = BeautifulSoup(page.content, "html.parser")
        all_publications = soup.find_all("div", class_="rendering_researchoutput_por-
tal-short")

        for publication in all_publications:
            title_element = publication.find("h3", class_="title")
            title = title_element.a.text.strip() if title_element and title_element.a
else "Title not available"
            publication_link = title_element.a["href"] if title_element and title_ele-
ment.a else "Link not available"

            authors = publication.find_all("a", class_="link person")
            authors_text = ", ".join(author.text.strip() for author in authors) if au-
thors else "Author not available"

            author_links = [author["href"] for author in publication.find_all("a",
class_="link person")]
            author_links_text = ", ".join(author_links) if author_links else "Author
links not available"

            publication_date_element = publication.find("span", class_="date")
            publication_date = publication_date_element.text.strip() if publica-
tion_date_element else "Publication Date not available"

            publication_obj = Publication(title, authors_text, author_links, publica-
tion_date, publication_link)
            self.publications.append(publication_obj)

    def crawl_and_parse(self):
        for page_num in range(self.num_pages):
            url = f"{self.base_url}?page={page_num + 0}"
```

```
            print(f"Fetching publications from page {page_num + 0}...")
            self.fetch_publications_from_page(url)

    def build_inverted_index(self):
        stemmer = PorterStemmer()
        stop_words = set(stopwords.words("english"))

        for n, publication in enumerate(self.publications, 1):
            # Construct the text content for stop-word removal and stemming
            text_content = f"{publication.title} {publication.authors} {publica-
tion.publication_date}"
            terms = word_tokenize(text_content.lower())

            filtered_text = []
            for term in terms:
                if term not in stop_words:
                    term = stemmer.stem(term)
                    filtered_text.append(term)

                    # Build the inverted index
                    if term in self.inverted_index:
                        self.inverted_index[term].append(n)
                    else:
                        self.inverted_index[term] = [n]

    def rank_documents(self, query_terms):
        matched_docs = {}
        for term in query_terms:
            if term in self.inverted_index:
                docs = self.inverted_index[term]
                for doc in docs:
                    matched_docs[doc] = matched_docs.get(doc, 0) + 1

        ranked_docs = sorted(matched_docs.items(), key=lambda x: x[1], reverse=True)
        return [doc_id for doc_id, _ in ranked_docs]

    def search_publications(self, query):
        query_terms = word_tokenize(query.lower())

        stemmer = PorterStemmer()
        stop_words = set(stopwords.words("english"))
        query_terms = [stemmer.stem(term) for term in query_terms if term not in
stop_words]

        matched_docs = self.rank_documents(query_terms)

        matched_publications = [self.publications[idx - 1] for idx in matched_docs]

        if not matched_publications:
            print("No publications found for the given query.")
            return

        print(f"Found {len(matched_publications)} publications matching the query
'{query}':\n")
        for publication in matched_publications:
            print(f"Publication:")
            print(f"Title: {publication.title}")
            print(f"Authors: {publication.authors}")
            if publication.author_links:
                print("Author Links:")
                for link in publication.author_links:
                    print(f"  {link}")
            print(f"Publication Date: {publication.publication_date}")
```

```
            print(f"Publication Link: {publication.publication_link}")
            print("-" * 40)

    def print_crawled_data(self):
        print("\nCrawled Publications:")
        for n, publication in enumerate(self.publications, 1):
            print(f"Publication {n}:")
            print(f"Title: {publication.title}")
            print(f"Authors: {publication.authors}")
            if publication.author_links:
                print("Author Links:")
                for link in publication.author_links:
                    print(f"  {link}")
            print(f"Publication Date: {publication.publication_date}")
            print(f"Publication Link: {publication.publication_link}")
            print("-" * 40)

    def print_inverted_index(self):
        print("\nInverted Index:")
        for term, docs in self.inverted_index.items():
            print(f"{term}: {docs}")
if __name__ == "__main__":
    base_url = "https://pureportal.coventry.ac.uk/en/organisations/centre-global-learn-
ing/publications/"
    num_pages = 6
    search_engine = PublicationSearchEngine(base_url, num_pages)
    search_engine.crawl_and_parse()
    search_engine.build_inverted_index()
    search_engine.print_crawled_data()
    search_engine.print_inverted_index()
    while True:
        user_query = input("Search For Publications or Authors (type 'exit' to quit):
")
        if user_query.lower() == "exit":
            break

        print("\nSearching for matching publications...")
        search_engine.search_publications(user_query
```