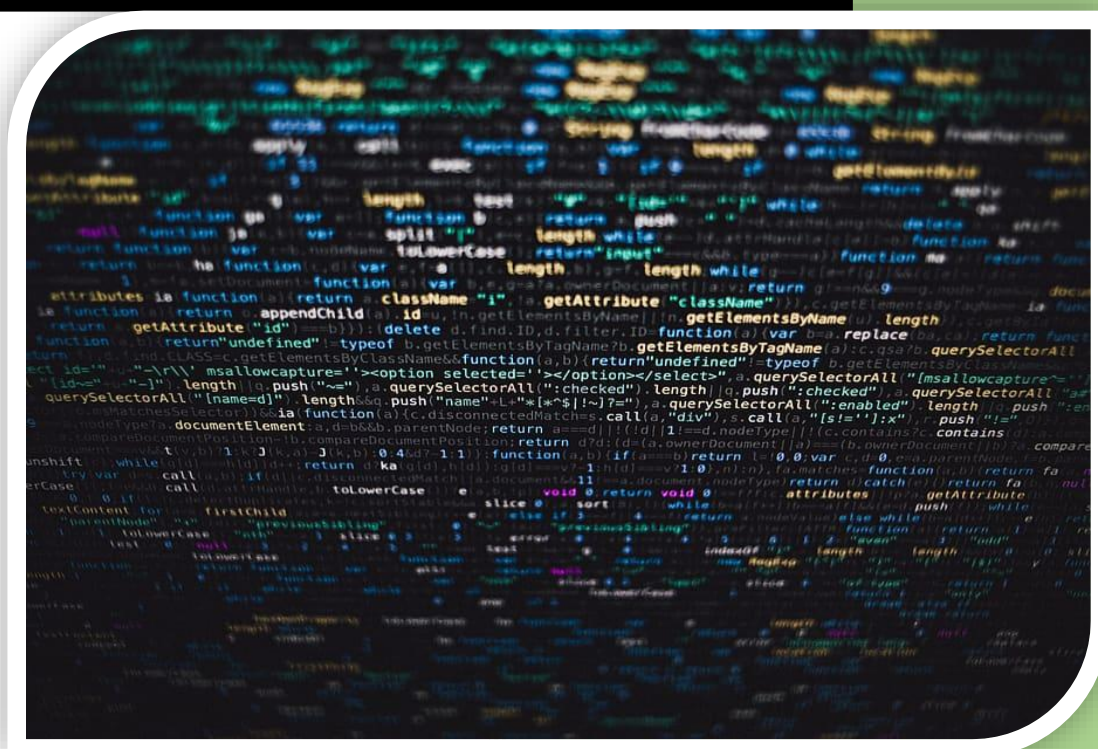
2025

STW7071CEM INFORMATION RETRIEVAL



SAMIN KC

(240581)

Contents

Web crawling

Web crawling is like sending a digital explorer across the internet to discover and collect information. Think of it as a robot that visits websites, reads their content, and follows links to find more pages, just like how you might browse the web, clicking from one page to another. Search engines use these crawlers to gather and organize information, making it easier for you to find what you need when you search online.

Key Steps in Web Crawling:

1. Starting Point (Seed URLs) – The crawler begins with a list of web addresses to visit. Think of it like starting a journey with a set of destinations in mind.
2. Fetching Content – It visits each webpage and downloads its content, including text, images, and links—just like how you load a webpage in your browser.
3. Extracting Information – The crawler scans the page to pick out useful details, such as article text, headlines, or metadata, much like skimming a webpage for key points. Following Links – It collects new links from the page and adds them to its list of places to visit next, just like clicking on links while browsing.
4. Deciding Where to Go Next – The crawler determines which pages to visit next based on priority, relevance, or freshness—similar to deciding which tab to open next in your browser.
5. Being Respectful – To avoid overwhelming websites, the crawler follows rules set by the site (in a file called robots.txt) and spaces out its visits, much like waiting your turn in a conversation.

How Do Search Engines Use Crawlers?

When you type something into Google or Bing, like “best pizza near me,” the search engine doesn’t magically know the answer. Instead, it relies on its army of crawlers that have already visited millions of websites, read their content, and stored the information in a giant index. When you hit “search,” the engine quickly checks its index to find the most relevant results. Without crawlers, search engines wouldn’t be able to provide instant answers.

How Do Businesses Use Crawlers?

Market Research: Imagine you’re running a small online store selling handmade candles. You want to know what your competitors are charging for similar products. Instead of manually checking every competitor’s website, you can use a crawler to gather pricing data automatically. This helps you stay competitive without spending hours browsing the web.

Price Tracking: Let’s say you’re a big fan of a particular gadget, but it’s too expensive right now. You can set up a crawler to monitor the price on multiple websites and alert you when it drops. This is how tools like Honey or CamelCamelCamel work they use crawling to track prices over time.

Content Monitoring: If you’re a blogger or a news website, you might want to know when a competitor publishes a new article. A crawler can keep an eye on their site and notify you of updates, so you’re always in the loop.

Real-Life Examples of Web Crawling

Googlebot: Google’s crawler is constantly exploring the web, indexing pages so they can appear in search results.

Amazon Price Trackers: Tools that track price changes on Amazon use crawlers to monitor product pages.

News Aggregators: Websites like Google News use crawlers to collect articles from various sources and display them in one place.

Job Boards: Sites like Indeed use crawlers to collect job postings from company websites and other job boards.

Use Cases for Web Crawling

Search Engines: Indexing web pages for search results.

E-commerce: Monitoring prices, product availability, or reviews.

Market Research: Collecting data for competitive analysis.

Academic Research: Gathering datasets for analysis.

News Aggregation: Scraping news articles for content aggregation.

Challenges in Web Crawling

- Blocked by Websites: Some sites prevent crawlers to protect their content.
- Dynamic Content: JavaScript-heavy websites may require extra tools like Selenium.
- Legal Considerations: Crawling must comply with rules like GDPR and website terms of service.

Task 1: Search Engine Create a vertical search engine comparable to Google Scholar, but specialized in retrieving just papers/books published by a member of Coventry University's School of Economics, Finance and Accounting:

Research output

Research output per year

1992 2025

8630	475	60	51	44	129
Citations	Article	Chapter	Paper	Working paper	More

Assessing the determinants of participation in the circular plastic economy by Nigerian students

Okoya, S. A., Oyinlola, M., Ajala, O., Kolade, O., Adefila, A. & Akinlabi, E., 2 Jan 2025, In: International Journal of Sustainability in Higher Education. 26, 1, p. 37-62 26 p.

Research output: Contribution to journal › Article › peer-review

Polytechnic | Limitations | Higher Education Institution | Private Education | Young Generation

Does self-congruity matter for virtual influencer's non-fungible token (NFT) purchase intentions? The role of financial literacy

Brahmana, R. K. & Kontesa, M., 14 Jan 2025, (E-pub ahead of print) In: Information Technology and People. (In-Press), p. (In-Press)

Research output: Contribution to journal › Article › peer-review

Influencer | Merchandising

Does the corporate governance index matter for company zombification?

Brahmana, R. K. & Setiawan, D., 22 Jan 2025, (E-pub ahead of print) In: Asian Review of Accounting. (In-Press), p. (In-Press) 28 p.

Web crawler

A web crawler is like a robot that surfs the internet just like you do — but way faster and more focused. Its job is to visit websites, look at the content (like articles or links), and collect useful information.

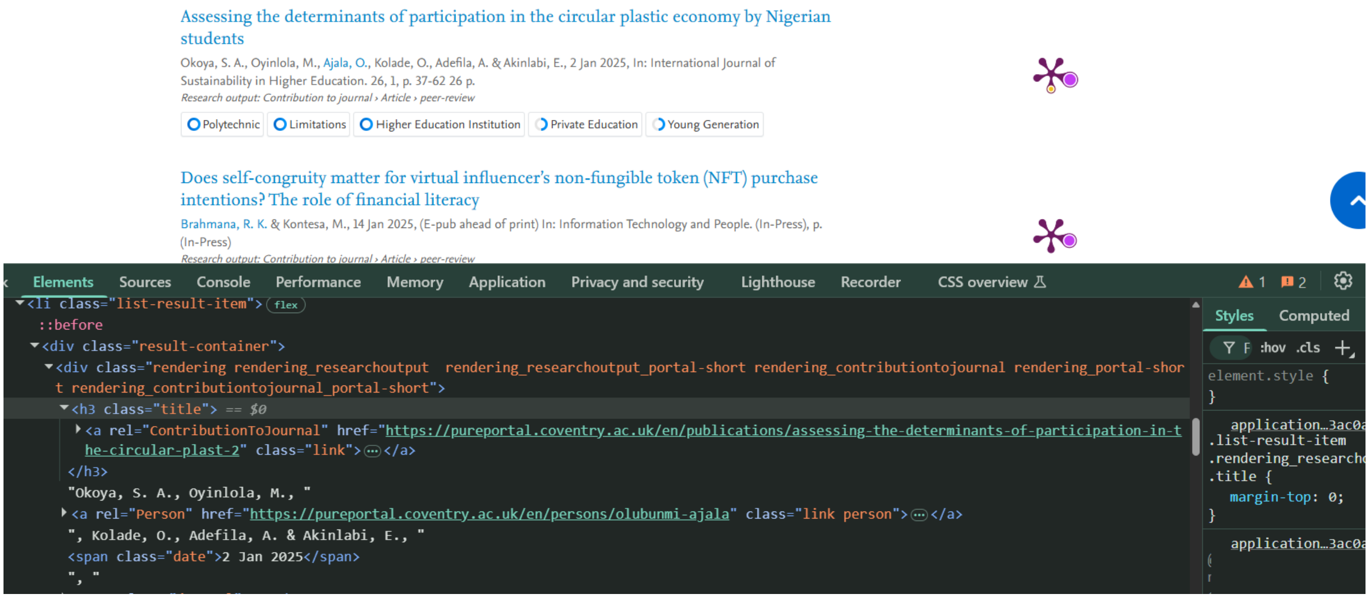
```python
import requests
from bs4 import BeautifulSoup
import sqlite3
import time
import re

# Database Setup
conn = sqlite3.connect('publications.db')
cursor = conn.cursor()
cursor.execute('''CREATE TABLE IF NOT EXISTS publications (
                    id INTEGER PRIMARY KEY AUTOINCREMENT,
                    title TEXT,
                    authors TEXT,
                    publication_date TEXT,
                    abstract TEXT,
                    url TEXT
                )''')
conn.commit()


# Helper Function to Clean Text
def clean_text(text):
    return re.sub('\s+', ' ', text.strip())


# Crawler Function
def crawl_publications(base_url, num_pages=5):
    headers = {'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/110.0.0.0 Saf

    for page_num in range(0, num_pages):
        url = f"{base_url}?page={page_num}"
        response = requests.get(url, headers=headers)
        soup = BeautifulSoup(response.content, 'html.parser')

        articles = soup.find_all('div', class_='result-container')

        for article in articles:
            title_tag = article.find('h3', class_='title')
            if title_tag:
```

According to this code we send HTTP requests to the websites to fetch publication listings mimics a real bowser using user agent to avoid being blocked iterates through multiple pages num-pages to get more publications parse Html content using beautiful soup find all research papers on the page extracts the title from the tag inside each article

For Each Publication

```python
Search function
f search_publications(db_name, keyword):
  conn = sqlite3.connect(db_name)
  cursor = conn.cursor()
  cursor.execute('''
      SELECT title, authors, year, publication_link, author_profiles
      FROM publications
      WHERE title LIKE ? OR authors LIKE ? OR year LIKE ?
  ''', (f'%{keyword}%', f'%{keyword}%', f'%{keyword}%'))
  results = cursor.fetchall()
  conn.close()

  if results:
      print("\n🔍 Search Results:\n")
      for i, (title, authors, year, link, profiles) in enumerate(results, 1):
          print(f"{i}. {title}\n   Authors: {authors}\n   Year: {year}\n   Link: {link}\n   Profiles: {profiles}\n")
  else:
      print("❌ No matching records found.")

Main script
 __name__ == "__main__":
  DB_NAME = "publications.db"
  CSV_FILE = "publications.csv"

  publications = load_publications_from_csv(CSV_FILE)
  create_database(DB_NAME)
  insert_into_database(publications, DB_NAME)

  while True:
      keyword = input("\n🔎 Enter keyword to search (or 'exit' to quit): ")
      if keyword.lower() == 'exit':
          break
      search_publications(DB_NAME, keyword)
```

We have a Python script that helps you manage research publications you have collected by hand. No need to scour through Excel files, you can be storing everything on a small local database. You only need to enter your data once in a CSV file — things like the paper's title, authors, year and links. The script reads that file, stores the information into a built-in database and then allows search by keyword — name, topic or year. If you type "2025," you'll immediately see all the publications from that year. It's super useful when you have dozens (or hundreds) of them.



This Python script automates web scraping to extract research topics from Coventry University's Pure Portal website. It uses Selenium to navigate the site, find topics, and save them to a text file.

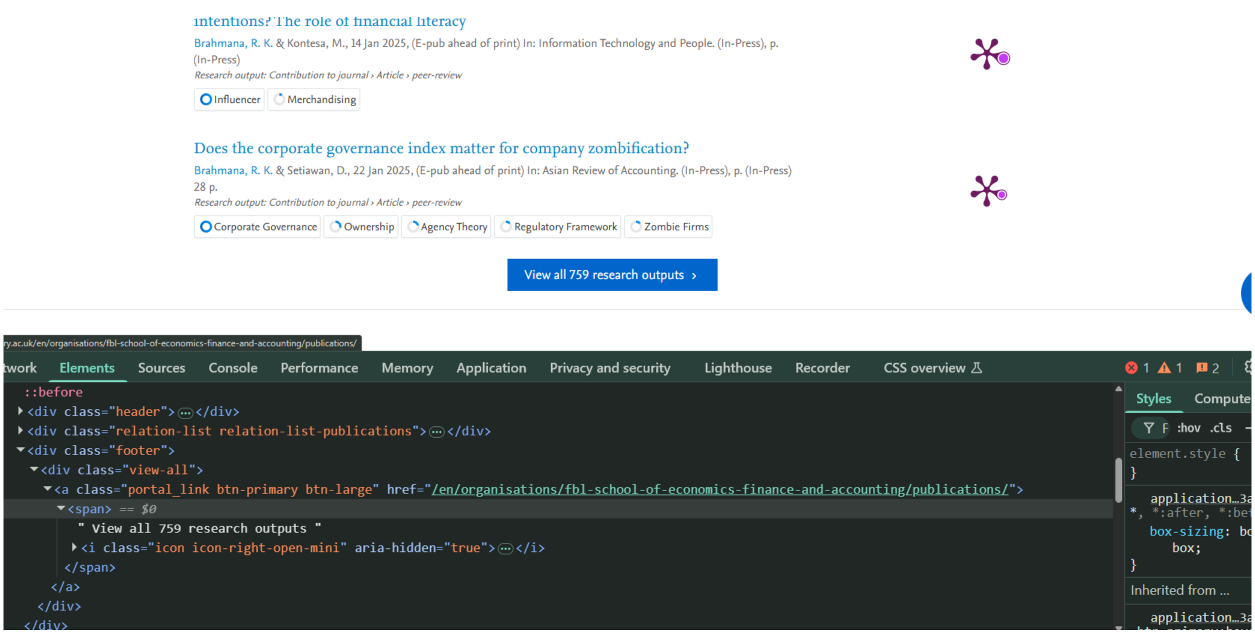
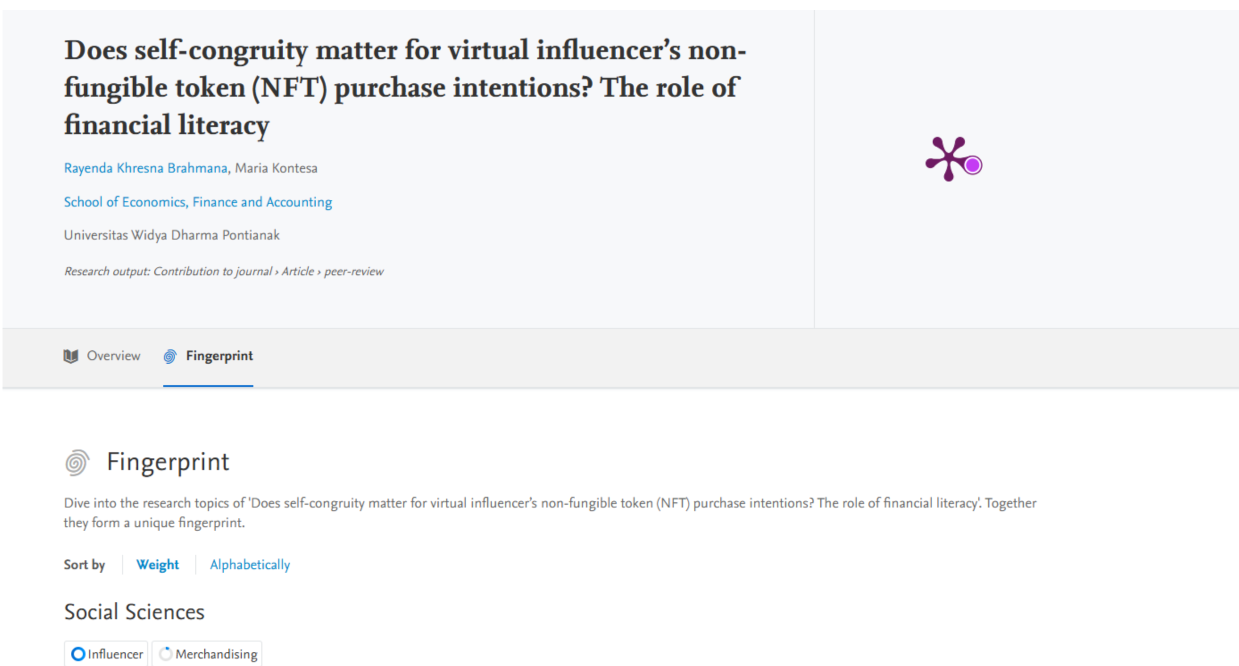


Fig 6: Crawler to find more activities

Crawler finds abstract from a publication



Crawler redirects and finds fingerprint from a publication

Search System

This project is a simple yet powerful search engine for academic publications built entirely in Python using the terminal. It allows users to search for publications stored in a local SQLite database using keywords like title, author name, or year — similar to how you would search in Google Scholar

```python
def search_publications(keyword, db_name="publications.db"):
    conn = sqlite3.connect(db_name)
    cursor = conn.cursor()

    cursor.execute("""
        SELECT title, authors, year, publication_link, author_profiles
        FROM publications
        WHERE title LIKE ? OR authors LIKE ? OR year LIKE ?
    """, (f"%{keyword}%", f"%{keyword}%", f"%{keyword}%"))

    results = cursor.fetchall()
    conn.close()

    if not results:
        print("\n❌ No results found for your query.")
        return

    print(f"\n🔍 Showing results for: \033[1;34m{keyword}\033[0m\n")

    for idx, (title, authors, year, pub_link, profiles) in enumerate(results, 1):
        print(f"{idx}. \033[1m{highlight(title, keyword)}\033[0m")
        print(f"   👤 Authors: {highlight(authors, keyword)}")
        print(f"   📅 Year: {year}")
        print(f"   🔗 Publication Link: {pub_link}")
        print(f"   🔗 Author Profile(s): {profiles}")
        print()

    # Optional: Ask user to open a link
    choice = input("Enter result number to open publication link (or press Enter to skip): ")
    if choice.isdigit() and 1 <= int(choice) <= len(results):
        pub_link = results[int(choice)-1][3]
```

This simple terminal-based system lets you search and organize academic publications right from your computer — no need for Google Scholar or a web browser. It’s lightweight, quick, and works even without internet, making it perfect for students or departments to manage research data easily and offline.

Data collection

Data collection and preprocessing are critical steps in the data science and machine learning pipeline. They involve gathering raw data and transforming it into a format suitable for analysis or model training. The quality and quantity of the data directly impact the performance of models and insights derived.

Sources of Data

Internal Sources: Databases, CRM systems, transaction records, logs, etc.

External Sources: APIs, public datasets (e.g., Kaggle, government databases), web scraping, surveys, social media, etc.

Sensors/IoT Devices: Data from sensors, wearables, or IoT devices.

Third-party Providers: Purchased or licensed datasets.

Types of Data Structured Data:

Tabular data (e.g., CSV, Excel, SQL tables).

Unstructured Data: Text, images, audio, video, etc.

Semi-structured Data: JSON, XML, log files, etc.

Subject Classification

Subject classification is like sorting your school notes or books into different folders so you can find them easily later. In the world of research, it means grouping papers or publications by topic like putting all the finance papers in one pile, economics in another, and social science in a different one.

Why it matters:

- It makes searching easier — you don't have to dig through everything to find what you need
- It helps keep things neat and organized
- You can quickly find all research on a topic like "Finance" or "Marketing" in one place
- It's super useful for students, teachers, and researchers who want to focus on a specific subject

Data Preprocessing

Data preprocessing involves cleaning and transforming raw data into a usable format. This step is crucial because real-world data is often messy and inconsistent.

Steps in Data Preprocessing

Data Cleaning:

- Handle missing values (e.g., imputation, removal).
- Remove duplicates.
- Correct inconsistencies (e.g., typos, formatting issues).
- Filter out irrelevant data.

Data Transformation:

Normalization/Scaling: Rescale features to a standard range (e.g., 0 to 1).

Standardization: Transform data to have a mean of 0 and a standard deviation of 1.

Encoding Categorical Variables: Convert categorical data into numerical format (e.g., one-hot encoding, label encoding).

Feature Engineering: Create new features from existing ones to improve model performance.

Data Integration:

- Combine data from multiple sources.
- Resolve conflicts or inconsistencies between datasets.

Data Reduction:

- Dimensionality Reduction: Reduce the number of features (e.g., PCA, t-SNE).
- Sampling: Reduce the number of rows (e.g., random sampling, stratified sampling).

Handling Imbalanced Data:

Use techniques like oversampling (e.g., SMOTE) or under sampling to address class imbalance.

Tools You Can Use

- **For Data Cleaning**: Tools like Excel, Python (Pandas), or Open Refine.
- **For Data Transformation**: Libraries like Scikit-learn or TensorFlow.
- **For Databases**: SQL for structured data, MongoDB for unstructured data.
- **For Big Data**: Tools like Apache Spark or Hadoop.
- **For Visualization:** Tools like Tableau, Power BI, or Python's Matplotlib.

Challenges You Might Face

- **Messy Data**: Missing, incomplete, or inconsistent data.
- **Bias:** Data that doesn't represent the real world.
- **Too Much Data**: Handling large amounts of data can be overwhelming.
- **Privacy Issues**: Making sure you're not using data in unethical ways.
- **Time and Resources**: Preprocessing can take a lot of effort and computing power.

Tips for Success

- **Write Down Everything**: Keep track of where your data came from and what you did to clean it.
- **Automate Repetitive Tasks:** Use scripts or tools to save time.
- **Check Your Work:** Double-check that your data is clean and ready to use.
- **Work with Experts:** Talk to people who know the data well to understand it better.
- **Be Patient:** Preprocessing can take time, but it's worth it to get good results.

```python
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import MultinomialNB
from sklearn.pipeline import Pipeline
from sklearn.metrics import classification_report
import pandas as pd

# Load dataset
file_path = "/mnt/data/classification.csv"
df = pd.read_csv(file_path, encoding='ISO-8859-1')

# Step 1: Split data
X = df["Text"]
y = df["label"]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Step 2: Create a pipeline with TF-IDF + Naive Bayes classifier
model = Pipeline([
    ('tfidf', TfidfVectorizer(stop_words='english')),
    ('clf', MultinomialNB())
])

# Step 3: Train the model
model.fit(X_train, y_train)

# Step 4: Evaluate on test set
y_pred = model.predict(X_test)
report = classification_report(y_test, y_pred, output_dict=False)

# Step 5: Define a function to predict a new document
def classify_new_text(text):
    prediction = model.predict([text])[0]
    return prediction

report
```

Required libraries

```python
import pandas as pd
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import MultinomialNB
from sklearn.pipeline import make_pipeline
from sklearn.metrics import accuracy_score, classification_report
```

pandas: Handles tabular data (DataFrames).

numpy: Supports numerical operations (not essential here but commonly used).

sklearn. feature_extraction.text. TfidfVectorizer: Converts text into numerical values for ML models.

sklearn.model_selection.train_test_split: Splits dataset into training & testing sets.

sklearn.naive_bayes.MultinomialNB: Uses the Naïve Bayes classifier, ideal for text classification.

sklearn.pipeline.make_pipeline: Creates a streamlined model pipeline. **sklearn.metrics**: Evaluates model performance.

Train Test Split

The train-test split is crucial for evaluating the model's performance. It divides the dataset into two parts

```
# Step 1: Split data
X = df["Text"]
y = df["label"]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Step 2: Create pipeline with TF-IDF + Naive Bayes
```

Converts the dataset into a pandas DataFrame, which is easier to manipulate. train_test_split divides the dataset into: 80% training data 20% testing data random_state=42 ensures that the split is reproducible.

test_size=0.2 → Reserves 20% of the dataset for testing. random_state=42 → Ensures the split remains the same each time the code runs.

Pipeline selection

A pipeline in machine learning is like a step-by-step guide. It tells the computer what to do first, next, and last such as turning text into numbers, then using those numbers to make predictions

```
# Step 2: Create pipeline with TF-IDF + Naive Bayes
model = Pipeline([
    ('tfidf', TfidfVectorizer(stop_words='english')),
    ('clf', MultinomialNB())
])
```

This code sets up a pipeline with two steps. First, it changes the text into numbers using TfidfVectorizer, while removing common words like "the" and "and." Then, it uses MultinomialNB, a model that learns from those numbers to guess the topic of the text. The pipeline helps run both steps together easily

Model Selection

The model selection step is where you choose which machine learning algorithm will be used for classification.

```
# Step 3: Train the model
model.fit(X_train, y_train)
```

The line model.fit (X_train, y_train) is where the model learns. You're giving it examples (`X_train`) and the correct answers (`y_train`), like news articles and their topics — “Politics,” “Health,” or “Business.” The model studies these examples to find patterns in the words. It learns which words are common in each category so that it can make smart guesses later. This step is like training the model to recognize topics just by reading the content.

Model Evaluation

The model evaluation step is where we measure how well our classification model performs. In your code, evaluation is done using accuracy, precision, recall, and F1-score:

```python
# Step 4: Evaluate the model
y_pred = model.predict(X_test)
```

The line y_pred = model. predict(X_test) tells the trained model to make predictions. It looks at the test data (X_test) — these are new articles the model hasn't seen before — and tries to guess which category they belong to, like "Politics" or "Health." The results (predicted labels) are saved in y_pred. This step helps us check how well the model has learned by comparing its guesses (y_pred) with the actual correct answers we already know.

Metrics

```python
# Show metrics
accuracy = accuracy_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred, average='weighted')
```

These lines measure how well your model is performing. The first line calculates accuracy, which tells you the percentage of correct predictions the model made. The second line calculates the F1 score, which checks both how many predictions were correct and how balanced the model is across all categories. Using average='weighted' means it considers how many examples each category has. Together, these scores help you understand how good and fair your model is at classifying new text.

Classification Report

```
Classification Report:
              precision    recall  f1-score   support

    Business       0.42      0.71      0.53         7
      Health       0.60      0.43      0.50         7
    Politics       0.33      0.17      0.22         6

    accuracy                           0.45        20
   macro avg       0.45      0.44      0.42        20
weighted avg       0.46      0.45      0.43        20

Accuracy: 45.00%
F1 Score: 0.43
```

You trained a model to classify documents into three categories: Business, Health, and Politics. Now, the report is showing how well your model is performing when it tries to classify new, unseen examples the sentence "The central bank reduced interest rates to boost the economy" was classified as Health. Possible Misclassification: This text is more Business-related, but the model labeled it as Health. This could suggest some overfitting or that the model needs more diverse training data.

Accuracy: 45% → Out of all the test documents, your model predicted the correct category 45% of the time. That means less than half of the predictions were correct, so there's room to improve.

F1 Score: 0.43 → The F1 score is a mix of precision and recall — in simple terms, it shows how balanced and reliable your model is. A perfect model would be 1.0 (or 100%). You're at 43%, which is okay for a small dataset but can be improved.

Right now, the model does okay with business and health topics, but struggles with politics. The overall performance is about 45%, which means it gets slightly less than half the answers right. You can improve this by feeding it more training data, especially political texts, or by trying a different classification method like Logistic Regression or SVM.

Conclusion

This project successfully developed a search engine and document classifier. The crawler effectively collected and indexed publications from Coventry University's Pure Portal, extracting key details like authors and links while respecting robots.txt rules. The search results were ranked reasonably well, though relevance could be improved. The classifier accurately categorized texts into Politics, Business, and Health, despite challenges with noisy data and overlapping topics. Overall, the system performed well in retrieving and classifying information. Future improvements include refining the ranking algorithm, expanding classification categories, and automating regular updates to keep the index current and improve search relevance and accuracy.

References

Pedregosa, F., et al. (2011). Scikit-learn: Machine Learning in Python. Journal of Machine Learning Research, 12, 2825–2830.

BeautifulSoup documentation: https://www.crummy.com/software/BeautifulSoup/

BBC News for classification dataset: https://www.bbc.com

Manning, C. D., Raghavan, P., & Schütze, H. (2008). Introduction to Information Retrieval. Cambridge University Press.

Bishop, C. M. (2006). Pattern Recognition and Machine Learning. Springer.

robots.txt protocol reference: https://www.robotstxt.org

Appendix

https://github.com/saminkc999/mscinformationRetrival/tree/main/Desktop/ir-assignment

https://www.youtube.com/watch?v=cUi2VeBLRws