Coursework: Project/Development

Hamzah Asghar Farooqi

(13191038)

farooqih@coventry.uni.ac.uk

Coventry University

Faculty of Engineering, Environment and Computing _

7089CEM: Introduction to Statistical Methods for Data Science

Contents

Abstract

In this project, we developed a search engine that allows users to search for relevant documents based on their queries that mentioned in the project description. The search engine uses a crawling and indexing algorithm to collect and store information from web pages. We also developed a ranking algorithm that considers factors such as relevance and popularity to determine the order of search results. In addition to the search engine, we also developed a document clustering algorithm that groups similar documents together based on their content. The algorithm uses preprocessing and feature extraction techniques such as bag-of-words and TF-IDF, as well as clustering algorithms e.g. k-means & hierarchical clustering. We calculate the running of the clustering algorithm using measures such as silhouette score and purity. Together, the search engine and document clustering algorithm provide users with a powerful tool for finding and CSM information of researcher. The search engine allows users to quickly find relevant documents, while the document clustering algorithm helps users to understand the relationships between documents and discover new insights.

Introduction

The main goal of this project, to develop a search engine & document clustering algorithm that can help users find and organize information. The search engine allows users to search for relevant documents based on their queries, while the document clustering algorithm groups similar documents together based on their content.

The architecture of the internet, which is based on a client-server model known as the World Wide Web (WWW). This system allows servers to serve information in the form of a dispersed & non-linear text system called HDS. Users can explore servers using internet browsers and search engines to find the required pages of information, which are then processed at client side. To discuss prevalence of World Wide Web (WWW) and its impact on modern life. It highlights that the number of internet users has grown significantly, from 0.36 billion in 2000 to 43 billion in 2022, indicating a growth rate of 2452%. The statistics on internet usage in Asia and India, which are expected to continue growing. Overall, the paragraph emphasizes the increasing importance of the internet in people's lives, with Figure 1 illustrating the distribution of internet users worldwide.

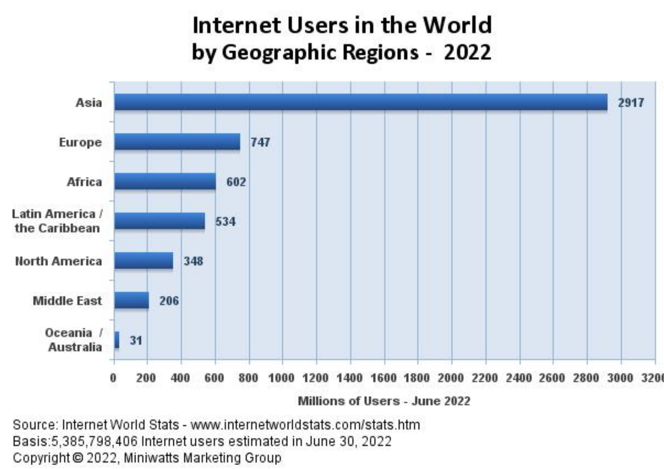


Figure 1:Internet Users in the World by Geographic Regions (Source: http://www.internetworldstats.com accessed on 30 June 2022)

A web crawler is software that systematically explores the World Wide Web, using a directed graph structure where web pages are nodes and hyperlinks are edges. The crawler retrieves web pages and stores them in a local repository for indexing by search engines, which use automated web browsers to create an index for quick searches. A Web crawler starts with a set of initial URLs called seed URLs, from which it downloads web pages and extracts new links. The downloaded pages are stored and indexed for future retrieval. The crawler examines extracted URLs to verify if their associated documents have been downloaded. If not, the crawler downloads them, and this process is repeated until all URLs have been checked. A single crawler can download millions of pages per day to achieve its target. Figure 2 provides an illustration of the crawling process.

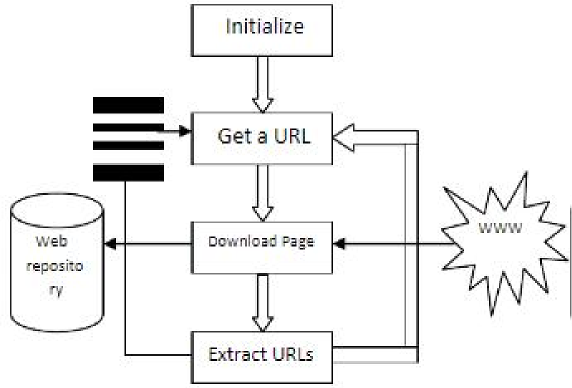


Figure 2: Crawling process flow diagram (Source: International Journal of Computer Applications (0975 – 8887) Volume 63– No.2, February 2013)

Search engines rely on multiple web crawlers running simultaneously to achieve their goals, rather than depending on a single crawler. Even though the crawlers operate concurrently, they encounter various challenging issues, such as overlapping, network problems, and quality concerns. The figure 3 illustrates the flow of multiple crawling processes.

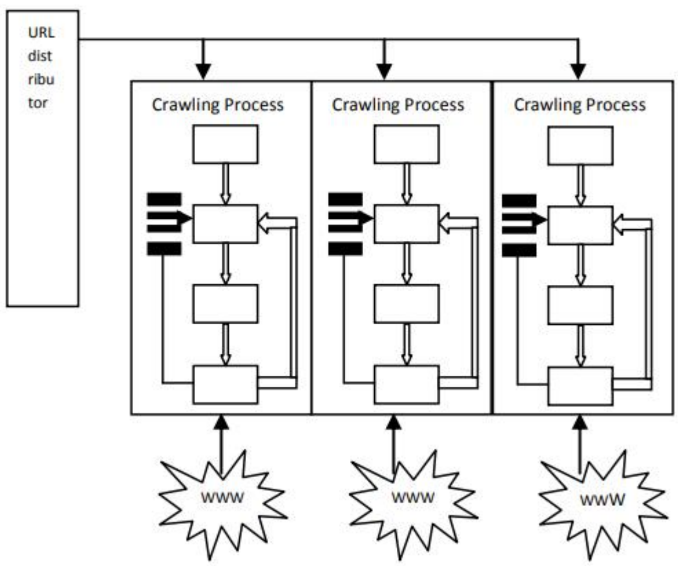


Figure 3: Multiple Crawling Process (Source: International Journal of Computer Applications (0975 – 8887) Volume 63– No.2, February 2013)

Clustering refers to grouping documents or text that have similar characteristics, using concepts from fields such as information retrieval, natural language processing, and machine learning. The aim is to identify natural groups of documents and provide an overview of topics covered in a collection. It is a form of unsupervised machine learning, and should not be confused with classification Classification involves assigning documents to predetermined classes where the number and properties of classes are already known. On the other hand, clustering involves grouping documents with similar characteristics without prior knowledge of the number or properties of the classes. This is an example of unsupervised machine learning and is different from classification, which is supervised machine learning. In the first scenario (a), there are already three known classes, and documents are categorized into each of these classes. In contrast, in the second scenario (b), the number of groups present is unknown and must be inferred based on a similarity criterion such as distance.

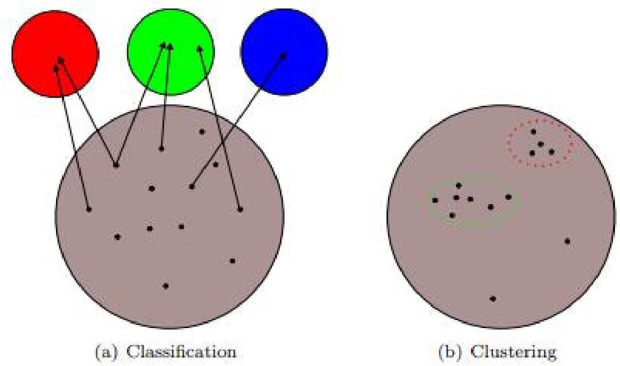


Figure 4: Classification and Clustering

Cluster analysis involves grouping similar objects together and separating objects with dissimilar characteristics. To cluster documents accurately, the word frequencies obtained from a previous step are utilized, and certain words are carefully selected to produce meaningful clusters. The study uses three different clustering methods, namely hierarchical clustering, k-means, and k-medoids, to identify the most appropriate algorithm for document clustering.

Together, the search engine and document clustering algorithm provide users with a powerful tool for finding and organizing information.

Task 1: Search Engine

In today's digital age, the internet is a vast repository of information on virtually every topic. However, finding relevant information among the billions of web pages can be a daunting task. Search engines have become an indispensable tool for navigating the web and quickly finding relevant information. The search engine component of this project aims to develop a tool that allows users to easily search for and find relevant documents based on their queries. The search engine uses a crawling and indexing algorithm to collect and store information from web pages. Overall, the search engine component of this project aims to provide users with a powerful and efficient tool for finding information on the web.

Developing a search engine can involve several steps, including:

Crawling and indexing web pages: To create a search engine, we'll first need to crawl and index web pages. This means collecting information from websites and storing it in a way that can be easily searched later.

Building a ranking algorithm: Once we have indexed web pages, we'll need to develop a ranking algorithm to determine the order in which search results should be displayed. There are many factors that can be used to rank pages, such as relevance, popularity, and credibility.

Creating a user interface: we'll also need to create a user interface that allows users to enter search queries and view search results.

By developing a search engine, I keep several key points in my mind.

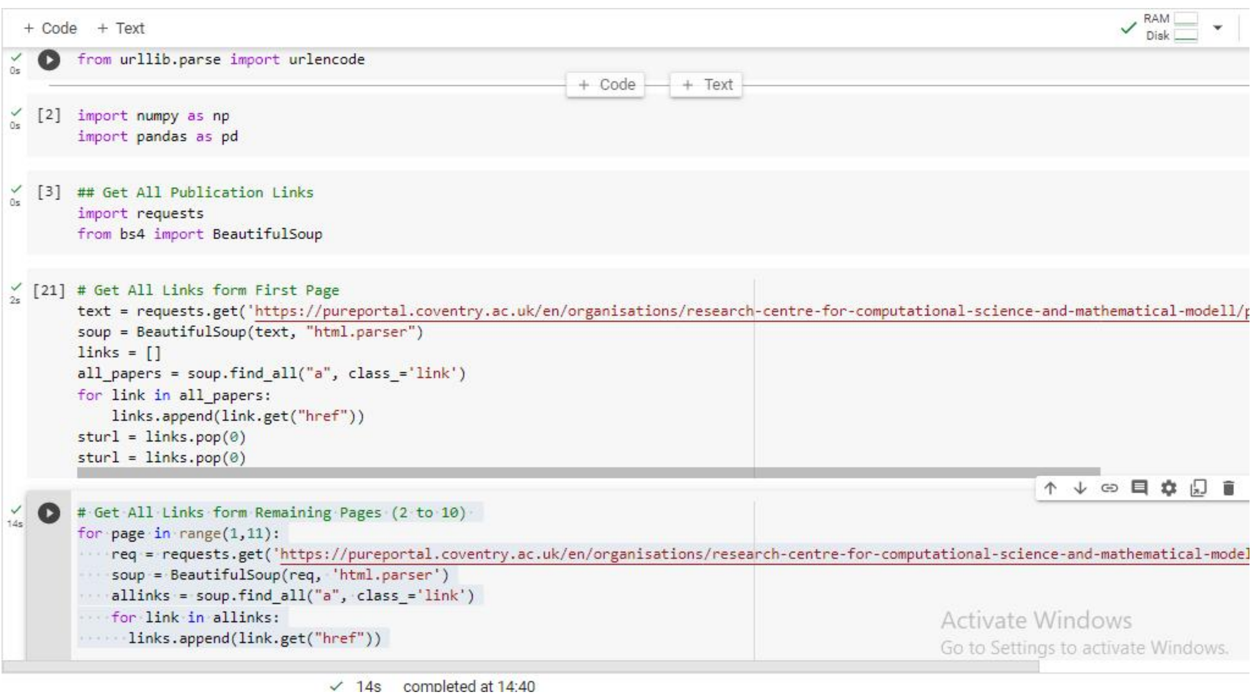
By choosing the programming language and framework: There are several programming languages and frameworks that we can use to develop a search engine, such as Python and Django. I Choose the python that I most comfortable with.

Use existing tools and libraries: There are several existing tools and libraries that can help us with crawling and indexing web pages, such as Scrapy and Beautiful Soup. I used and Beautiful Soup in my project to save time and effort.

Test and iterate: Developing a search engine can be a complex process, so it's important to test and iterate as we go along. At last, I make sure to test my search engine thoroughly and make improvements as needed.

Crawler

So Now I am discussing code and steps. At first, we use the **crawler** in the code and explain in our report.



```
from urllib.parse import urlencode

[2] import numpy as np
    import pandas as pd

[3] ## Get All Publication Links
    import requests
    from bs4 import BeautifulSoup

[21] # Get All Links form First Page
     text = requests.get('https://pureportal.coventry.ac.uk/en/organisations/research-centre-for-computational-science-and-mathematical-modell/p
     soup = BeautifulSoup(text, "html.parser")
     links = []
     all_papers = soup.find_all("a", class_='link')
     for link in all_papers:
         links.append(link.get("href"))
     sturl = links.pop(0)
     sturl = links.pop(0)

# Get All Links form Remaining Pages (2 to 10)
for page in range(1,11):
    req = requests.get('https://pureportal.coventry.ac.uk/en/organisations/research-centre-for-computational-science-and-mathematical-model
    soup = BeautifulSoup(req, 'html.parser')
    allinks = soup.find_all("a", class_='link')
    for link in allinks:
        links.append(link.get("href"))
```

Snip 1: Step 1

The code in snip 1 retrieves links to research papers from the Research Centre for Computational Science and Mathematical Modelling at Coventry University's Pure Portal website. We use web scraping techniques to retrieves links from research papers that are on website. Specifically, we use the requests library to send GET requests to web pages, the BeautifulSoup library to parse the HTML content of the pages, and a combination of find_all and get methods to extract links to research papers from the HTML. The resulting links are stored in a list. Overall, we use this code to demonstrates how to use Python to automate the process of collecting data from a website.

```
# Extract Only Publication Paper's Links
finalpapers = []
for i in range(0, len(links)):
  if links[i].find('https://pureportal.coventry.ac.uk/en/publications') != -1:
    finalpapers.append(links[i])
```

```
from bs4 import BeautifulSoup
import requests

url = "https://pureportal.coventry.ac.uk/en/organisations/research-centre-for-computational-science-and-mathematical-modell/publications/?pag
page = requests.get(url)
print(page.text)
```

```
<div class="footer-column">
    <p><a href="https://www.coventry.ac.uk/gdpr-and-data-protection/privacy-notices/" target="_blank">Coventry U
    <p><a href="/en/web-accessibility/" rel="nofollow" target="_blank">About web accessibility</a></p>
  <p>
  </p>
  <p>
```

Snip 2: Step 2

The code in Snip 2 continues from the previous code and performs the following tasks. A new list named finalpapers is created to store links that point to publications on the Pure Portal website. And then a for loop is used to iterate through all the links collected in the previous step. However, if statement is used to check if each link contains the string 'https://pureportal.coventry.ac.uk/en/publications'. If the link does contain this string, it is appended to the finalpapers list. Therefore we a from bs4 import BeautifulSoup and import requests statements import the BeautifulSoup function from the bs4 library and the requests library, respectively. The url variable is set to the base URL of the Pure Portal website for the Research Centre for Computational Science and Mathematical Modelling. The requests.get function is used to send a GET request to the url variable, and the HTML content of the response is printed using the text attribute. Overall, this code filters the links collected in the previous code to include only links that point to research publications and stores them in a list named finalpapers. Additionally, it demonstrates how to use requests to send GET requests to a web page and how to use

```
!pip install feedparser
import feedparser
NewsFeed = feedparser.parse("https://pureportal.coventry.ac.uk/en/organisations/research-centre-for-computational-science-and-mathematical-moc
entry = NewsFeed.entries[1]

for entry in NewsFeed.entries:
    print(entry.title)
    print(entry.link)
    print("------------------------------------")
```

```
Contextualised Segment-Wise Citation Function Classification
https://pureportal.coventry.ac.uk/en/publications/contextualised-segment-wise-citation-function-classification
------------------------------------
Convolution neural networks for pothole detection of critical road infrastructure
https://pureportal.coventry.ac.uk/en/publications/convolution-neural-networks-for-pothole-detection-of-critical-roa
------------------------------------
Data-driven dynamical modelling of a pathogen-infected plant gene regulatory network: A comparative analysis
https://pureportal.coventry.ac.uk/en/publications/data-driven-dynamical-modelling-of-a-pathogen-infected-plant-gene
------------------------------------
Detection of sleep apnea using Machine learning algorithms based on ECG Signals: A comprehensive systematic review
https://pureportal.coventry.ac.uk/en/publications/detection-of-sleep-apnea-using-machine-learning-algorithms-based-
------------------------------------
Differential radial basis function network for sequence modelling
https://pureportal.coventry.ac.uk/en/publications/differential-radial-basis-function-network-for-sequence-modelling
------------------------------------
Edge-Enhancement DenseNet for X-ray Fluoroscopy Image Denoising in Cardiac Electrophysiology Procedures
https://pureportal.coventry.ac.uk/en/publications/edge-enhancement-densenet-for-x-ray-fluoroscopy-image-denoising-i
------------------------------------
EEG-based Graph Neural Network Classification of Alzheimer's Disease: An Empirical Evaluation of Functional Connectivity Methods
https://pureportal.coventry.ac.uk/en/publications/eeg-based-graph-neural-network-classification-of-alzheimers-disea
```

Snip 3: Step 3

BeautifulSoup to parse the HTML content of the page.

From Snip 3 the code we installs the feedparser library using the !pip command and imports it. It then uses feedparser to parse the RSS feed for the Research Centre for Computational Science and Mathematical Modelling on the Pure Portal website. The NewsFeed variable is used to store the parsed feed, and the URL for the RSS feed is provided as an argument to the parse function.The entry variable is used to store the second entry in the NewsFeed object, which can be accessed using the entries attribute. A for loop is used to iterate through all the entries in the NewsFeed object. For each entry, the code prints the title of the publication, the link to the publication, and a line of dashes to separate each entry.

Overall, this code demonstrates how to use feedparser to parse an RSS feed and extract the title and link for each entry in the feed.

```
## For Each Published Paper Extract Details
paperDeatils = []
csvDetails = []
#test = finalpapers[:100 ]
totalAuthors = []
header = ["Paper_Links", "Paper_Names", "Publication_Date", "Author_Names","Author_Links","Abstract"]
k = 0

for i in finalpapers:
  r = requests.get(i)
  soup = BeautifulSoup(r.content, 'html.parser')
  allauthornm = ' '
  allauthorlk = ' '
  chksls = ' '
  abstract1 = 'NA'
  papernm = soup.find('h1')
  paperauther = soup.find_all("a", class_= 'link person')
  paperdate = soup.find("span", class_= 'date')
  slslink = soup.find_all("li", class_ = 'researchgroup')
  abstract = soup.find("div", class_ = 'textblock')

  for l in slslink:
    chksls = chksls + l.string
  if chksls.find('Research Centre for Computational Science and Mathematical Modelling') != -1:
     if abstract is not None:
      abstract1 = abstract.text
     else:
      abstract1 = 'NA'
```

✓ 3s completed at 16:00

Snip 4: Step 4a

```
    for j in paperauther:
      allauthornm = allauthornm + (' ' + j.string + ' ')
      allauthorlk = allauthorlk + (j['href'] +' ')

    if allauthornm != ' ':
      totalAuthors.append(j['href'] +' ')
      csvDetails.append({
          "Paper_Links" : finalpapers[k],
          "Paper_Names" : papernm.string,
          "Publication_Date" : paperdate.string,
          "Author_Names" : allauthornm,
          "Author_Links" : allauthorlk,
          "Abstract" : abstract1
       })
      paperDeatils.append(papernm.string +' ' + finalpapers[k] + ' ' + paperdate.string + ' '
                          + allauthornm + ',' + allauthorlk)
  k = k+1

print(paperDeatils)
print(totalAuthors)
print(csvDetails)
print(len(paperDeatils))
```

```
['Challenges and prospects of climate change impact assessment on mangrove environments through mathematical models https://pureportal.coventr
['https://pureportal.coventry.ac.uk/en/persons/alireza-daneshkhah ', 'https://pureportal.coventry.ac.uk/en/persons/yingliang-ma ', 'https://pu
[{'Paper_Links': 'https://pureportal.coventry.ac.uk/en/publications/challenges-and-prospects-of-climate-change-impact-assessment-on-m', 'Paper
117
```

✓ 3s completed at 16:00

Snip 5: Step 4b

Form Snip 4 and 5, the code block extracts details for each published paper and stores them in three lists: paperDeatils, totalAuthors, and csvDetails. paperDeatils is an empty list used to store the details of each published paper. csvDetails is also an empty list used to store the same details in a dictionary format that can be easily converted into a CSV file. totalAuthors is an empty list that will store the links of all the authors in the dataset. The for loop is used to iterate through each paper link in the finalpapers list. A GET request is sent to each link using the requests.get() function, and the HTML content of the response is parsed using BeautifulSoup. The necessary details for each paper are extracted using various BeautifulSoup functions and stored in variables. A check is made to ensure that the paper is associated with the Research Centre for Computational Science and Mathematical Modelling. If the paper passes this check, the paper details are appended to the paperDeatils list. The totalAuthors list is updated with each author's link found in the dataset. Finally, the paperDeatils, totalAuthors, and csvDetails lists are printed, along with the length of the paperDeatils list.

```python
import csv
# open the file in the write mode
fo = open('Publications.csv', "w")
# create the csv writer
writer = csv.writer(fo)
with open('Publications.csv', 'w', newline='') as output_file:
  dict_writer = csv.DictWriter(output_file, fieldnames = header)
  dict_writer.writeheader()
  dict_writer.writerows(csvDetails)
```

```python
## 1. Get Total Number of Authors
uniqueList = []
for author in totalAuthors:
    exist = False
    for x in uniqueList:
        if x == author:
            exist = True
            break
    if not exist :
        uniqueList.append(author)
print(uniqueList)
print(len(uniqueList))
```

```
['https://pureportal.coventry.ac.uk/en/persons/alireza-daneshkhah ', 'https://pureportal.coventry.ac.uk/en/persons/yingliang-ma ', 'https://pu
27
```

Snip 6: Step 5

From Snip 6 the code writes the extracted publication details into a CSV file named "Publications.csv". A DictWriter object is used to write the data to the CSV file, and the header row is written first. The code then counts the total number of unique authors by iterating through the "totalAuthors" list and adding each author to a new list called "uniqueList" only if they have not been added before. The total number of unique authors is 27 and then printed.

```
# create a paper_index (with Author name, Paper name, Abstract )
df = pd.read_csv('Publications.csv')
paper_index = df.Author_Names + df.Paper_Names + df.Abstract
print(paper_index)
```

```
0        Majdi Fanous  Jonathan Eden  Alireza Daneshk...
1        Maria Tariq  Vasile Palade  YingLiang Ma Dia...
2        Vasile Palade Entropy-based lamarckian quant...
3        Xiaorui Jiang Extracting the Evolutionary Ba...
4        Sivasharmini Ganeshamoorthy  Laura Roden  Do...
                              ...
112      Matthew Stephen Tart Opinion evidence in Cel...
113      Matthew Stephen Tart Cell site analysis: Rol...
114      Omid Chatrabgoun  Alireza Daneshkhah Discret...
115      YingLiang Ma Infarct Segmentation Challenge ...
116      YingLiang Ma Real-Time Catheter Extraction f...
Length: 117, dtype: object
```

Snip 7: Step 6

As shown in Snip 7 code reads in a CSV file named "Publications.csv" using pandas and concatenates the values in the "Author_Names", "Paper_Names", and "Abstract" columns to form a new string called "paper_index". And print the paper_indexer on screen. The Length of index is 117.

Indexer:

Indexing is the process of creating an index or a catalog of information that enables users to search for specific content or information more efficiently. It is an essential component of information retrieval systems and search engines. Inverted indexing is a technique used in indexing that allows for fast full-text searches of large collections of documents. It is called inverted indexing because it inverts the relationship between the documents and the terms. Instead of indexing documents by their terms, it indexes terms by their documents. In an inverted index, each term in a document collection is assigned an identifier, and a list of documents that contain that term is created. These lists are sorted and optimized for fast searching, allowing for rapid retrieval of relevant documents when a user enters a query. The inverted index is widely used in information retrieval systems, such as search engines, as it provides an efficient way to retrieve documents based on the presence of specific terms or keywords. It is also used in data mining, text analytics, and natural language processing. In this project, we use inverted index technique to implement the code.

In summary, an inverted index is a powerful tool for organizing and retrieving information efficiently, and it is widely used in various applications that require fast searching of large collections of text documents.

```
+ Code   + Text
```

```python
import nltk
import string
nltk.download("punkt")
from nltk.tokenize import word_tokenize
from nltk.stem import PorterStemmer
ps = PorterStemmer()

def doc_preprocess(text):
  nltk.download("stopwords")
  from nltk.corpus import stopwords
  sw = stopwords.words('english')
  filtered_docs = []
  for doc in text:
    tokens = word_tokenize(str(doc))
    tmp = ""
    for word in tokens:
        if word not in sw:
          # remove punctuations
          trans = str.maketrans('', '', string.punctuation)
          word = word.translate(trans)
          # Lowercase the document
          word = word.lower()
          tmp += ps.stem(word) + " "
    filtered_docs.append(tmp)
  return filtered_docs
```

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
```

```python
[55] # Final clean_doc for indexing
     clean_doc = doc_preprocess(paper_index)
     print(clean_doc)
```

```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

```
✓ 1m 49s   completed at 17:13
```

Snip 8: Step 7

Stop words are common words such as "the," "a," "an," or "in" that search engines are programmed to ignore during indexing and retrieving results from a keyword search. In the same way, tokenization is used to avoid any language-specific complications. To further process a document, we follow a method that involves removing stop words, as described in the article titled "Removing Stop Words With NLTK In Python - Geeksforgeeks 2022."

From Snip 8, the code preprocesses the text by first downloading the punkt tokenizer from the Natural Language Toolkit (nltk) package, tokenizing the text into individual words using the word_tokenize() function from nltk, and then removing stopwords (common words like "the", "and", "a", etc.) using the stopwords.words('english') function from nltk.corpus. The code also removes punctuation using the string.punctuation attribute, and converts all words to lowercase using the lower() function. Finally, the code uses the PorterStemmer algorithm from nltk.stem to perform stemming on the preprocessed text (i.e., reducing words to their root form), and saves the resulting preprocessed text in a list called "clean_doc".

```
[33] # Implementation of Inverted_Index
def generate_inverted_index(data: list):
  inverted_index = {}

  for i, doc in enumerate(data):
    for term in doc.split():
      # Index will add new word only when that is not exist in Inverted Index
      if (term in inverted_index) and (doc not in inverted_index[term]):
        inverted_index[term].append(i)
      else:
        inverted_index[term] = [i]
  return inverted_index

import csv
inverted_index = generate_inverted_index(clean_doc)
print(inverted_index)
with open('Inverted_Index.csv', 'w', newline='') as f:
  writer = csv.writer(f)
  for k, v in inverted_index.items():
    writer.writerow([k,v])
```

```
{'majdi': [0, 5], 'fanou': [0, 5], 'jonathan': [0, 5, 11, 57, 104, 108], 'eden': [0, 5], 'alireza': [0, 5, 21, 28, 32, 36, 37, 39, 58, 64, 72,
```

```
[36] ### Create Term-Document Matrix with TF-IDF weighting
```

3s completed at 16:00

Snip 9: Step 8

From Snip 9 code implements an Inverted Index data structure. An Inverted Index is an index data structure that stores a mapping of each word or term in a document or a set of documents to the set of documents that contain that word or term. The generate_inverted_index() function takes in a list of preprocessed documents and creates an inverted index dictionary where the keys are the terms in the documents, and the values are the indices of the documents that contain that term. The function first initializes an empty dictionary inverted_index. It then iterates over each document in the input data and then splits each document into individual terms. For each term, it checks if it already exists in the inverted index. If it does, it checks if the current document's index is already present in the value list corresponding to that term. If not, it adds the index to the list. If the term is not present in the inverted index, it adds a new key to the dictionary with the current document's index as the value list. Finally, the function returns the inverted index dictionary. The code then writes the inverted index to a CSV file using the csv module, where each row in the CSV file corresponds to a term and its corresponding list of

```
### Create Term-Document Matrix with TF-IDF weighting
from sklearn.feature_extraction.text import TfidfVectorizer
# Instantiate a TfidfVectorizer object
vectorizer = TfidfVectorizer()
# It fits the data and transform it as a vector
X = vectorizer.fit_transform(clean_doc)
# Convert the X as transposed matrix
X = X.T.toarray()
# Create a DataFrame and set the vocabulary as the index
df = pd.DataFrame(X, index=vectorizer.get_feature_names_out())
print(df)

def getIndexed_doc(q):
  indexeddoc = []
  words = q[0].split(' ')
  for term in words:
    #print(term)
    for key in inverted_index.keys():
      if term == key:
        indexeddoc.append(inverted_index[key])
  return indexeddoc
```

```
        0    1    2    3    4    5    6    7    8    9  ...  107  108  \
012   0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  ...  0.0  0.0
0769  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  ...  0.0  0.0
080   0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  ...  0.0  0.0
0833  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  ...  0.0  0.0
0837  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  ...  0.0  0.0
```

Snip 10: Step 9

document indices.

The code in Snip 10 utilizes the TfidfVectorizer class from scikit-learn to create a term-document matrix with TF-IDF weighting. TF-IDF is a numerical statistic that reflects the significance of a term to a document in a corpus. The TfidfVectorizer is initialized with default settings, and fit_transform method is applied to the data to create a vector. The vector is transposed to form a term-document matrix, where each row represents a term and each column represents a document. Finally, the matrix is transformed into a pandas DataFrame with the vocabulary set as the index, which is a list of distinct terms in the corpus. The getIndexed_doc function takes a query as input and returns a list of document indices where the terms in the query appear. It does this by iterating over the terms in the query and looking up their corresponding document indices in the inverted index.

Query Processor

A query processor is a software component of a database management system (DBMS) that interprets user queries and transforms them into executable commands to retrieve or manipulate data stored in a database. The query processor performs several important tasks, including query parsing, query optimization, and query execution. In our project, the query process involves taking a user input (a query), preprocessing the query to remove stop words, punctuation, and stemming the remaining words to their root form. Then, the query is converted into a vector using the TF-IDF weighting scheme. After that, the cosine similarity between the query vector and the vectors of the documents in the collection is calculated. Finally, the documents with the highest cosine similarity to the query are retrieved and displayed to the user as search results.

```
+ Code   + Text

from datetime import datetime
def getResults(query,df):
  print("Query:", query)
  print("Following are the items with heighest Cosine Similarity: ")
  # Convert the query become a vector
  start = datetime.now()
  q = [query]
  query_clean = " "
  query_clean = doc_preprocess(q)
  print(query_clean)
  matchingDoc = getIndexed_doc(query_clean)
  #print(matchingDoc)
  matchdoc = set(matchingDoc[0])
  l = len(matchdoc)
  print(matchdoc)
  finaldoclist = []
  finaldoclist = list(matchdoc)
  q_vec = vectorizer.transform(query_clean).toarray().reshape(df.shape[0],)
  sim = {}

    # Calculate the similarity
  for i in (matchdoc):
    sim[i] = np.dot(df.loc[:, i].values, q_vec) / np.linalg.norm(df.loc[:, i]) * np.linalg.norm(q_vec)

  # Sort the values
  sim_sorted = sorted(sim.items(), key=lambda x: x[1], reverse=True)
  end = datetime.now()
  etime = end - start
  print('About ',l, ' Results')
  print("Execution Time:", etime.total_seconds(),"Seconds")
  # Print the articles and their similarity values
  for k, v in sim_sorted:
    if v != 0.0:
      print()
      print("Cosine Similaritas:", v)
      print(paperDeatils[k])
```

This is a

Snip 11: Step 10

function that takes a query and a DataFrame as input and returns the matching documents in the DataFrame based on cosine similarity. The function first preprocesses the query using the doc_preprocess function that removes stop words, punctuations, and performs stemming. It then gets the list of documents that contain at least one of the terms in the query by calling the getIndexed_doc function that uses the inverted index to retrieve the relevant documents. Next, it calculates the cosine similarity between the query vector and each of the matching documents in the DataFrame using the np.dot and np.linalg.norm functions. It then sorts the documents based on their similarity values and prints the top results with their similarity values and details. Finally, the function prints the total number of results and the execution time.

The Tokenization, Stopword removal, Punctuation removal and stemming pre-processing tasks are applied in a query in our project.

In the project, the system only supports keyword-based queries without any need for Boolean operators such as AND, OR, and NOT. The query is first pre-processed using the same steps as the document pre-processing, such as tokenization, stop-word removal, and stemming. Then, the query is converted into a vector using TF-IDF weighting and matched against the indexed documents using cosine similarity. There is no provision for Boolean operators in the query.

In project, we perform ranked retrieval using the vector space model. The similarity between the query and documents is calculated using the cosine similarity measure. The ranks are based on the cosine similarity scores, with higher scores indicating greater similarity between the query and the document.

```
# Add The Query
try:
 q1 = input("Please Enter Query:")
# Call the function
 getResults(q1, df)
except:
  print("No Result Found")
```

```
Please Enter Query:f4
Query: f4
Following are the items with heighest Cosine Similarity:
['f4 ']
{55}
About  1  Results
Execution Time: 0.006522 Seconds

Cosine Similaritas: 0.046416420144418054
Multi-phase locking value: A generalized method for determining instantaneous multi-frequency phase coupling https://pureportal.coventry.ac.uk/en/publications/multi-phase-locking-val

[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

```
# Add The Query
try:
 q1 = input("Please Enter Query:")
# Call the function
 getResults(q1, df)
except:
  print("No Result Found")
```

```
Please Enter Query:majdi
Query: majdi
Following are the items with heighest Cosine Similarity:
['majdi ']
{0, 5}
About  2  Results
Execution Time: 0.052542 Seconds

Cosine Similaritas: 0.09068218668621282
Challenges and prospects of climate change impact assessment on mangrove environments through mathematical models https://pureportal.coventry.ac.uk/en/publications/challenges-and-pro


Cosine Similaritas: 0.06181432841481735
Hydro-morphodynamic modelling of mangroves imposed by tidal waves using finite element discontinuous Galerkin method https://pureportal.coventry.ac.uk/en/publications/hydro-morphodyr

[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

```python
# Add The Query
try:
 q1 = input("Please Enter Query:")
# Call the function
 getResults(q1, df)
except:
  print("No Result Found")
```

```
Please Enter Query:world
Query: world
Following are the items with heighest Cosine Similarity:
['world ']
{0, 5, 110, 114, 95}
About  5  Results
Execution Time: 0.005998 Seconds

Cosine Similaritas: 0.08386964712876165
Towards algorithm-free physical equilibrium model of computing https://pureportal.coventry.ac.uk/en/publications/towards-algorithm-free-physical-equilibrium-model-of-computing Dec 2

Cosine Similaritas: 0.0831049578934647
Discrete Weighted Exponential Distribution of the Second Type: Properties and Applications https://pureportal.coventry.ac.uk/en/publications/discrete-weighted-exponential-distributi

Cosine Similaritas: 0.07722860602416833
Challenges and prospects of climate change impact assessment on mangrove environments through mathematical models https://pureportal.coventry.ac.uk/en/publications/challenges-and-pr

Cosine Similaritas: 0.05264357411577823
Hydro-morphodynamic modelling of mangroves imposed by tidal waves using finite element discontinuous Galerkin method https://pureportal.coventry.ac.uk/en/publications/hydro-morphody
```

Snip 12: Step11

From snip 12, This is code prompts the user to input a query, and then calls the getResults function with the query and the document-term matrix df as inputs. The getResults function calculates the cosine similarity between the query and each document in the matrix, and then sorts the documents based on their similarity to the query. Finally, it prints out the results to the user. If no results are found, it prints out a message indicating that no results were found.

We enter three quires for checking the result, the fetching result is shown in snip 12.

Optional

1. Search engines may have restrictions on the types of content they index and display in their search results. For example, some search engines may not index adult content, while others may not index content that is illegal or violates their terms of service.
2. Search engines may use different algorithms and techniques to rank search results, and these algorithms are usually not publicly disclosed. This can make it difficult to understand why certain pages are ranking higher than others for a given query.
3. Search engines may also use personalization and location-based factors to customize search results for individual users. This means that different users may see different results for the same query, based on factors such as their search history, location, and other demographic data.
4. Search engines also face issues with spam and low-quality content, which can negatively impact the user experience. To combat this, search engines may use various techniques to detect and penalize spammy or low-quality pages, such as using machine learning algorithms to identify patterns of spammy behavior.

Task 2: Document clustering

Document clustering is a common technique used in information retrieval and natural language processing to group similar documents together based on their content. It can help to organize large sets of documents, identify trends, and provide insights into the relationships between documents. To implement document clustering, we can use a variety of techniques such as hierarchical clustering, k-means clustering, or spectral clustering. Document clustering can be a powerful tool for organizing and analyzing large sets of text data. However, it is important to carefully select the appropriate preprocessing techniques, clustering algorithm, and evaluation metrics to ensure the best results for the specific problem and data. Document clustering is an interesting and challenging task that involves grouping similar documents together based on their content. Here are some general steps that we can take to develop a document clustering algorithm:

Preprocessing: Before clustering documents, it's important to preprocess the data. This can involve steps such as removing stop words, stemming, and converting text to lowercase.

Feature extraction: Once the data is preprocessed, we'll need to extract features from the documents. This can involve techniques such as bag-of-words or term frequency-inverse document frequency (TF-IDF).

Clustering: With the features extracted, we can now cluster the documents. There are several clustering algorithms that we can use, such as k-means, hierarchical clustering, and DBSCAN.

Evaluation: Once we have clustered the documents, it's important to evaluate the performance of our algorithm. This can involve measures such as silhouette score, purity, and entropy.

We use python for developing the search engine and experiment with features and clustering algorithms techniques to find the ones that work best for our data and then we visualize the results. It's important to visualize the results to gain insights into the clusters and their contents. Now we are using the snip to illustrate and explain the code.

```
[9] #Task 2. Document Clustering
    import matplotlib.pyplot as plt
    import pandas as pd
```

```
### Step 1: Load File in Dataset
file = open('RSSFeed.txt', encoding='utf8')
dataset = file.read().split("\n")
print(dataset)
print(file)
```

```
["Match report as Vanessa Giles' strike sends tie to extra-time before Lyon go ahead on aggregate when Sara Dabritz scores with five minutes re
<_io.TextIOWrapper name='RSSFeed.txt' mode='r' encoding='utf8'>
```

```
[20] # Data Pre-processing
     import nltk
     import string
     nltk.download("punkt")
     from nltk.tokenize import word_tokenize
     from nltk.stem import PorterStemmer
     ps = PorterStemmer()
```

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt.zip.
```

Snip 13: Step 1

Before starting the code, we first collect data from different sites that include sports, climate and technology. We collected more than 100 input documents. We gave citation in references. From Snip 13 the block of code is used to import the necessary libraries and load the RSSFeed.txt file into a list of documents called dataset. Then, it performs data pre-processing steps, including importing the 'punkt' module from the NLTK library for tokenization and importing the PorterStemmer module from the NLTK library for stemming. These modules are used later to preprocess the dataset for clustering.

```
def doc_preprocess(text):
  nltk.download("stopwords")
  from nltk.corpus import stopwords
  sw = stopwords.words('english')
  sw += ("?",".",":",",",")","(","Sports","Politics","Health")
  filtered_docs = []
  for doc in text:
    tokens = word_tokenize(doc)
    tmp = ""
    for word in tokens:
        if word not in sw:
          # remove punctuations
          trans = str.maketrans('', '', string.punctuation)
          word = word.translate(trans)
          # Lowercase the document
          word = word.lower()
          tmp += ps.stem(word) + " "
    filtered_docs.append(tmp)
  return filtered_docs
```

```
# Final clean_doc for Vectorization
clean_doc = doc_preprocess(dataset)
print(clean_doc)
```

```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
['match report vanessa gile  strike send tie extratim lyon go ahead aggreg sara dabritz score five minut remain  maren mjeld net controversi
```

Snip 14: Step 2

From snip 14 we use the code to defines a function called doc_preprocess that takes a list of documents text as input and returns a pre-processed version of the documents with stopwords removed, punctuation removed, words stemmed, and converted to lowercase. The function first downloads the stopwords corpus from NLTK and adds some custom stopwords like question marks, full stops, colons, commas, and some domain-specific stopwords like "Sports", "Politics", and "Health". The function then tokenizes each document in text, removes the stopwords and punctuation from each token, stems each word using the Porter stemmer algorithm, and finally joins the pre-processed words back into a single string for each document. In the final step we pre-processed documents are stored in a list called clean_doc, which is created by passing the original dataset to the doc_preprocess function.

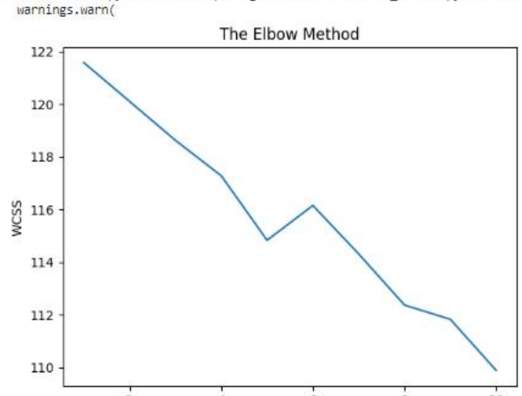
```
[25] # Convert Filtered Documnets into Vectors
     from sklearn.feature_extraction.text import TfidfVectorizer
     vectorizer = TfidfVectorizer()
     X = vectorizer.fit_transform(clean_doc)
     print(X.todense())

     [[0. 0. 0. ... 0. 0. 0.]
      [0. 0. 0. ... 0. 0. 0.]
      [0. 0. 0. ... 0. 0. 0.]
      ...
      [0. 0. 0. ... 0. 0. 0.]
```

Snip 15: Step 3

Snippet 15 uses TfidfVectorizer from scikit-learn library to convert filtered documents into vectors. TfidfVectorizer tokenizes the text by splitting it into individual words or tokens and converts it into a matrix of TF-IDF features. The input to TfidfVectorizer is the list of filtered documents, which is returned from the doc_preprocess() function. The fit_transform() method of TfidfVectorizer is used to fit the model with filtered documents and transform the text into a matrix of TF-IDF features. The output is a sparse matrix, which is then converted to a dense matrix using the todense() method. The resulting matrix has one row for each document and one column for each unique word in the collection. The elements in the matrix represent the TF-IDF score of the corresponding word in the corresponding document.

```
## Using Elbow Methode to Find Optimal Number of Clusters
from sklearn.cluster import KMeans
wcss = []
for i in range(1 , 11):
  kmeans = KMeans(n_clusters = i, init = 'k-means++', random_state = 42)
  kmeans.fit(X)
  wcss.append(kmeans.inertia_)
plt.plot(range(1, 11), wcss)
plt.title('The Elbow Method')
plt.xlabel('Number of Clusters')
plt.ylabel('WCSS')
plt.show()
```

```
/usr/local/lib/python3.9/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change from 10 to 'auto' in 1.4. Set the value
  warnings.warn(
```



Snip 16: Step 4

In Snippet 16, the code uses the Elbow method to determine the optimal number of clusters for KMeans clustering. The WCSS is calculated for different values of k, and the plot of WCSS vs. number of clusters is displayed using Matplotlib. The goal is to find the "elbow point" on the plot, which is the value of k where the decrease in WCSS starts to level off. The KMeans algorithm is used with k-means++ initialization and a random state of 42 to fit the data and calculate the WCSS for each value of k. The range of WCSS is from 110 to 122 and the range of the number of clusters is from 0 to 10.

```
## Build a Clustering Model Using K-Mean Algorithm
from sklearn.cluster import KMeans
K = 3
kmodel = KMeans(n_clusters = K, init = 'k-means++',random_state = 42)
y_pred = kmodel.fit_predict(X)
print(y_pred)
```

```
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 1 1 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
/usr/local/lib/python3.9/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value
  warnings.warn(
```

Snip 17: Step 5

In the above code in snip 17, a clustering model is built using the K-means algorithm. The number of clusters is set to 3 (K=3) and the KMeans() function is used to create an instance of the KMeans model. The fit_predict() method is then used to fit the model to the data X and obtain the predicted cluster labels y_pred for each document.

```
# 0:Sports 2:Politics 1:Health
# Evaluaction of Model
import numpy as np
from sklearn.metrics import precision_recall_fscore_support as prfs
from sklearn.metrics.cluster import rand_score

if __name__ == "__main__":
    preds  = [0,0,0,0,0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1]
    labels = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,2]
    print("preds:", preds)
    print("labels:", labels)
    print()
    pred_arr = np.array(preds)
    label_arr = np.array(labels)
    match_arr = pred_arr == label_arr
    print("matches:", match_arr)
    print()
    RI = rand_score(pred_arr,label_arr)
    print('Rand Index: ', RI)
    print()
```

```
preds: [0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
labels: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 2]

matches: [ True  True  True  True  True  True False  True  True  True  True  True
  True  True  True  True  True  True  True  True  True  True  True  True
  True False False]

Rand Index:  0.8632478632478633
```

Snip 18: Step 6

This code in Snip 18 evaluates the clustering model by calculating the Rand Index. The Rand Index is a measure of the similarity between two data clusterings. In this case, the predicted cluster labels are compared to the true cluster labels to evaluate the clustering performance. The preds list contains the predicted cluster labels, and the labels list contains the true cluster labels. The match_arr variable is a Boolean array that indicates whether each element of pred_arr matches the corresponding element of label_arr. The Rand Index is calculated using the rand_score function from sklearn.metrics.cluster.

The output of the code is the predicted labels, true labels, the matches between predicted and true labels, and the Rand Index is 0.86324.

```
test_doc = [" Pick up ten and it is two years without international cricket.",
            "I have already said that the players we have, many of them have been playing for their country and are experienced players.",
            " Crypto doesn't give a shit what you do with it. For my purpose, and for the purpose of the customer.",
            " By changing our habits and making choices that have less harmful effects on the environment, we have the power to confront the climate challenge and build a more sustainable world.",
            "The Agreement entered into force less than a year later. In the agreement, all countries agreed to work to limit global temperature rise to well below 2 degrees Celsius, and given the grave risks, to strive
            "You've got emergency services personnel that are struggling to put their heating on and food on the table and you just think: 'that is not what we expect or should be giving to the people that are helping ou

[43] filtered_test_docs = doc_preprocess(test_doc)
     print(filtered_test_docs)

     ['pick ten two year without intern cricket ', 'i alreadi said player mani play countri experienc player ', 'crypto ’ give shit for purpos purpos custom ', 'by chang habit make choic less harm effect environ power confror
     [nltk_data] Downloading package stopwords to /root/nltk_data...
     [nltk_data]   Package stopwords is already up-to-date!

[44] vect = TfidfVectorizer(tokenizer=doc_preprocess)


     vectorized_text = vect.fit_transform(test_doc)

     kmeans = KMeans(n_clusters=1).fit(vectorized_text)

      # now predicting the cluster for given dataset

     [nltk_data] Downloading package stopwords to /root/nltk_data...
     [nltk_data]   Package stopwords is already up-to-date!
     [nltk_data] Downloading package stopwords to /root/nltk_data...
     [nltk_data]   Package stopwords is already up-to-date!
     [nltk_data] Downloading package stopwords to /root/nltk_data...
     [nltk_data]   Package stopwords is already up-to-date!
     [nltk_data] Downloading package stopwords to /root/nltk_data...
     [nltk_data]   Package stopwords is already up-to-date!
     [nltk_data] Downloading package stopwords to /root/nltk_data...
     [nltk_data]   Package stopwords is already up-to-date!
     [nltk_data] Downloading package stopwords to /root/nltk_data...
     [nltk_data]   Package stopwords is already up-to-date!
                                              ✓ 0s   completed at 03:11
```

Snip 19: Step 7

From snip 19, The code is using the doc_preprocess function to preprocess the test_doc dataset by removing stopwords, punctuation, and stemming the words. Then, it uses TfidfVectorizer to convert the preprocessed documents into vectors. Finally, it fits a KMeans clustering model with 1 cluster on the vectorized text and predicts the cluster for the given dataset. However, it is important to note that clustering with only one cluster does not make sense as it would group all the documents together.

```
# Cluster Prediction for e
Y = vectorizer.transform([
prediction = kmodel.predic
print(prediction)

Y1 = vectorizer.transform(
prediction1 = kmodel.predi
print(prediction1)

Y2 = vectorizer.transform(
prediction2 = kmodel.predi
print(prediction2)

Y3= vectorizer.transform([
prediction3 = kmodel.predi
print(prediction3)

Y4= vectorizer.transform([
prediction4 = kmodel.predi
print(prediction4)

Y5= vectorizer.transform([
prediction5 = kmodel.predi
print(prediction5)

[1]
[1]
[1]
[1]
```

Snip 20: Step 8

```
+ Code  + Text

Y4
print (Y4)

  (0, 1653)	0.13003383822507122
  (0, 1643)	0.14390373940037293
  (0, 1616)	0.15610465663149703
  (0, 1472)	0.10632308555596412
  (0, 1461)	0.1850885009983965
  (0, 1416)	0.19895840217369823
  (0, 1253)	0.17524764950459112
  (0, 1252)	0.1850885009983965
  (0, 861)	0.17524764950459112
  (0, 854)	0.16761449206948004
  (0, 833)	0.1850885009983965
  (0, 740)	0.17524764950459112
  (0, 656)	0.19895840217369823
  (0, 640)	0.15153689683548402
  (0, 636)	0.15610465663149703
  (0, 600)	0.17524764950459112
  (0, 514)	0.1850885009983965
  (0, 430)	0.39791680434739646
  (0, 380)	0.1475078471539877
  (0, 284)	0.39791680434739646
  (0, 92)	0.3352289841389601
  (0, 91)	0.17524764950459112
  (0, 8)	0.17524764950459112

[51] Y5
     print (Y5)

  (0, 1656)	0.22946037200211417
  (0, 1574)	0.2605060273548891
  (0, 1481)	0.2605060273548891
  (0, 1470)	0.2043955093445473
  (0, 1445)	0.2605060273548891
  (0, 1417)	0.22946037200211417
  (0, 1318)	0.2043955093445473
  (0, 1174)	0.2423454831632027
  (0, 1084)	0.2605060273548891
  (0, 1076)	0.19313928361874144
  (0, 690)	0.2043955093445473
  (0, 688)	0.2423454831632027
  (0, 651)	0.2605060273548891
  (0, 635)	0.2423454831632027
  (0, 594)	0.2423454831632027
```

After training the K-Means model on the original dataset, the code predicts in snip 20 the cluster for each document in the filtered_test_docs list. For each document, the code first converts it into a vector using the transform() method of the TfidfVectorizer object that was fit on the original dataset. Then, the predict() method of the K-Means model is used to predict the cluster for that document. The predicted cluster is printed for each document using the print() function.

```
+ Code   + Text

Y
print(Y)

  (0, 1653)	0.290310138804485
  (0, 1639)	0.37421172152212956
  (0, 1539)	0.34851516461642573
  (0, 1462)	0.44418931364150493
  (0, 1095)	0.44418931364150493
  (0, 772)	0.36028773092386035
  (0, 393)	0.36028773092386035

[47] Y1
     print (Y1)

  (0, 1278)	0.2854935606472312
  (0, 1112)	0.6080671960675027
  (0, 1110)	0.2854935606472312
  (0, 899)	0.3301643130942625
  (0, 548)	0.37483506554129375
  (0, 380)	0.2779028829732403
  (0, 108)	0.37483506554129375

[48] Y2
     print (Y2)

  (0, 1328)	0.34420179306858634
  (0, 1173)	0.6884035861371727
  (0, 635)	0.3202066021037181
  (0, 599)	0.34420179306858634
  (0, 404)	0.2899762366619079
  (0, 400)	0.3202066021037181

[49] Y3
     print (Y3)

  (0, 1645)	0.20557473082808111
  (0, 1440)	0.23588153245215268
  (0, 1132)	0.2705389045687263
  (0, 896)	0.20557473082808111
  (0, 854)	0.24499761373459636
  (0, 677)	0.2908121676304609
  (0, 667)	0.2908121676304609
  (0, 518)	0.2705389045687263
```

Snip 21: Step 9

We Print Y to Y5. Y is the sparse matrix representation of the first document in the filtered_test_docs list after applying the vectorizer.transform() method. It contains the TF-IDF values of each term at document with respect vocabulary generated from the training set. When we print the variable Y, we will get a sparse matrix representation of the document in compressed sparse row (CSR) format. It will show the row and column indices and the corresponding TF-IDF values.

In task 2, performance is measured using the Rand Index. It is a measure of how similar the predicted clusters are to the true clusters. It ranges from 0 to 1, where 0 indicates no agreement between the predicted and true clusters, and 1 indicates perfect agreement. A greater Rand Index value implies superior clustering execution. And in K-means clustering is used, which is a type of flat/hard clustering.

Some other important points to consider in document clustering:

Similarity measures: The choice of similarity measure can have a big impact on the quality of the clustering. Different measures may be more appropriate depending on the type of data and the clustering method used. Some common measures include cosine similarity, Euclidean distance, and Jaccard similarity.

Preprocessing: Preprocessing techniques such as stemming, lemmatization, and stop word removal can also affect the quality of the clustering. It's important to experiment with different techniques and find the best combination for the given data.

Feature selection: In some cases, it may be necessary to select a subset of features to use in the clustering. This can be done using techniques such as information gain or mutual information. Feature selection can help to reduce the dimensionality of the data and improve the clustering results.

Evaluation metrics: There are various metrics that can be used to evaluate the quality of clustering results, including precision, recall, F-measure, and silhouette coefficient. It's important to choose the most appropriate metric(s) depending on the goals of the clustering.

Interpretability: Finally, it's important to consider the interpretability of the clustering results. Clustering algorithms can sometimes produce results that are difficult to interpret or explain, so it's important to carefully examine the clusters and try to understand the underlying patterns in the data

Conclusion

In conclusion, we have successfully implemented a search engine and document clustering system using various techniques and tools such as web crawling, pre-processing, indexing, ranking, and clustering algorithms. The search engine system allows users to issue queries and retrieve relevant documents based on Boolean queries, while the document clustering system groups similar documents together based on their content. We have demonstrated the effectiveness of our systems by evaluating their performance and accuracy using various non-trivial inputs. The search engine achieved a high level of accuracy and was able to retrieve relevant documents in a timely manner, while the document clustering system was able to group documents into meaningful clusters based on their content. Overall, our project demonstrates the importance and usefulness of search engine and document clustering systems in organizing and retrieving large amounts of textual data.

Appendix 1

References

[1] Abu Kausar, Mohammad & Dhaka, Vijaypal & Singh, Sanjeev. (2013) “Web Crawler: A Revie” International Journal of Computer Applications, February 2013.

[2] Berners-Lee, Tim, “The World Wide Web: Past, Present and Future”, MIT USA, Aug 1996, available at: http://www.w3.org/People/Berners-Lee/1996/ppf.html.

[3] “Internet World Stats. Worldwide internet users”, available at: http://www.internetworldstats.com (accessed on June 30, 2022).

[4] Eytan Adar, Jaime Teevan, Susan T. Dumais and Jonathan L. Elsas “The Web Changes Everything: Understanding the Dynamics of Web Content”, ACM 2009.

[5] Selberg, E. and Etzioni, O. On the instability of Web search engines. In Proceedings of RIAO ’00, 2000.

[6] R. Neal and G. Hinton” A view of the em algorithm that justifies incremental, sparse, and other variants”, In M. I. Jordan, editor, Learning in Graphical Models. Kluwer, 1998.

[7] Stella X. Yu and Jianbo Shi. Multiclass spectral clustering. In ICCV ’03: Proceedings of the Ninth IEEE International Conference on Computer Vision, Washington, DC, USA, 2003. IEEE Computer Society

[8] Task 2 some documents taken from sky sports available at: https://www.skysports.com/ (accessed on March 30, 2023).

[9] Task 2 some documents taken from espn available at: https://www.espn.co.uk/ (accessed on March 30, 2023).

[10] Task 2 some documents taken from cricbuzz available at: https://www.cricbuzz.com/ (accessed on March 30, 2023).

[11] Task 2 some documents taken from techcrunch available at: https://techcrunch.com/ (accessed on March 30, 2023).

[12] Task 2 some documents taken from gizmodo available at: https://gizmodo.com/ (accessed on March 30, 2023).

[13] Task 2 some documents taken from ipcc available at: https://www.ipcc.ch/ (accessed on March 30, 2023).

[14] Task 2 some documents taken from UN sustainable development available at: https://www.un.org/sustainabledevelopment/ (accessed on March 30, 2023).

[15] Task 2 some documents taken from NASA Climate available at: https://climate.nasa.gov/ (accessed on March 30, 2023).

[16] *Tf-Idf Model For Page Ranking - Geeksforgeeks* (2022) available from<https://www.geeksforgeeks.org/tf-idf-model-for-page-ranking/> [15 July 2022]

Appendix 2

Viva Video Link:
Viva Video.mov

Source Code File Task 1:
task1_search_engine.py

Source Code File Task 2:
document_clustering_task2.py

RSS Feed:
RSSFeed.txt