Appendix 1. Items covered in your report

**Part 1 – Search engine**

1. **Crawler:**
2. **All libries and source code used in crawling the publications website:**

1.1 Number of staff whose publications are crawled (approximately) and the maximum

number of publications per staff

**LIBRARIES AND FUNCTIONS:**

#!pip install PySimpleGUI

#!pip install scrapy

#!pip install requests beautifulsoup4

#!pip install nltk

#!pip install requests beautifulsoup4

import time

import requests

from bs4 import BeautifulSoup

import nltk

from nltk.corpus import stopwords

from nltk.tokenize import word\_tokenize

import bs4 as bs

import urllib.request

import scrapy

from datetime import date

1.2. Information collected about each publication (e.g. links, title, year, author or any additional part)

1.3.url = "https://pureportal.coventry.ac.uk/en/publications/"

response = requests.get(url)

soup = BeautifulSoup(response.content, "html.parser")

def research\_publication(url):

pagecrawled = 1

researchpublication = []

while True:

url = f"{url}/publications/?page={numberofpage}"

publisheddata = get\_publications\_data(url)

if not publisheddata:

break

all\_publications.extend(publications\_data)

crawled\_page += 1

time.sleep(1) # Add a delay of 1 second to be polite in crawling

return research\_publication

for publication in soup.select(".result-container"):

title = publication.select\_one(".title span").text

authors = [author.text for author in publication.select(".link.person span")]

publisher = publication.select\_one(".link[rel='Publisher'] span")

publication\_status = publication.select\_one(".date").text

publisher = publication.select\_one(".link[rel='Publisher'] span")

publication\_link = publication.select\_one(".title a")["href"]

for author in authors:

staff.add(author)

print(f"Title: {title}")

print(f"Authors: {', '.join(authors)}")

print(f"Publisher: {publisher}")

print(f"Publication status: {publication\_status}")

print(f"publication page Link: {link}")

print(f"CGLauthor: https://pureportal.coventry.ac.uk/en/persons/{author.lower().replace(' ', '-')}")

print()

1.4 NUMBER OF STAFF WHOSE PUBLICATION ARE CRAWLED:

url = "https://pureportal.coventry.ac.uk/en/publications/"

response = requests.get(url)

soup = BeautifulSoup(response.content, "html.parser")

staff = set()

for publication in soup.select(".result-container"):

title = publication.select\_one(".title span").text

authors = [author.text for author in publication.select(".link.person span")]

publisher = publication.select\_one(".link[rel='Publisher'] span")

publication\_status = publication.select\_one(".date").text

publisher = publication.select\_one(".link[rel='Publisher'] span")

publication\_link = publication.select\_one(".title a")["href"]

for author in authors:

staff.add(author)

#print(f"Title: {title}")

#print(f"Authors: {', '.join(authors)}")

#print(f"Publisher: {publisher}")

#print(f"Publication status: {publication\_status}")

#print(f"publication page Link: {link}")

#print(f"CGLauthor: https://pureportal.coventry.ac.uk/en/persons/{author.lower().replace(' ', '-')}")

#print()

print(f"Number of staff whose publications are crawled: {len(staff)}")

#print()

1.5 MAXIMUM NUMBER OF STAFF PUBLICATION:

url = "https://pureportal.coventry.ac.uk/en/publications/"

response = requests.get(url)

soup = BeautifulSoup(response.content, "html.parser")

staff = {}

max\_publications = 0

for publication in soup.select(".result-container"):

title = publication.select\_one(".title span").text

authors = [author.text for author in publication.select(".link.person span")]

publisher = publication.select\_one(".link[rel='Publisher'] span")

publication\_status = publication.select\_one(".date").text

publisher = publication.select\_one(".link[rel='Publisher'] span")

link = publication.select\_one(".title a")["href"]

for author in authors:

if author not in staff:

staff[author] = 0

staff[author] += 1

if staff[author] > max\_publications:

max\_publications = staff[author]

print(f"Maximum number of publications per staff: {max\_publications}")

print()

1.3. **Which pre-processing tasks are performed before passing data to Indexer:** :

To enhance the website functionality by adding additional line of code to be retrieved from the research publication web page and also removal of stop words. The added line checks if the title of the publication contains the string “Bletchley Park” and prints a message if it does and if the publisher of the publication is McGill-Queen’s University Press and prints a message if it found.

import requests

from bs4 import BeautifulSoup

url = "https://pureportal.coventry.ac.uk/en/publications/"

response = requests.get(url)

soup = BeautifulSoup(response.content, "html.parser")

staff = set()

for publication in soup.select(".result-container"):

title = publication.select\_one(".title span").text

authors = [author.text for author in publication.select(".link.person span")]

publisher = publication.select\_one(".link[rel='Publisher'] span")

publication\_status = publication.select\_one(".date").text

publication\_link = publication.select\_one(".title a")["href"]

for author in authors:

staff.add(author)

(f"Title: {title}")

(f"Authors: {', '.join(authors)}")

(f"Publisher: {publisher}")

(f"Publication status: {publication\_status}")

(f"publication Link: {link}")

(f"CGLauthor: https://pureportal.coventry.ac.uk/en/persons/{author.lower().replace(' ', '-')}")

(f"Number of staff whose publications are crawled: {len(staff)}")

# Add additional functionality here

#for title in publication:

for publication in soup.select(".result-container"):

title = publication.select\_one(".title span").text

if "Bletchley Park" in title:

print(f"Found a publication about Bletchley Park: {title}")

if "McGill-Queen's University Press":

print(f"Found a publication by McGill-Queen's University Press: {title}")

**preprocessing before indexing: removal of stop words from author and title**

url = "https://pureportal.coventry.ac.uk/en/publications/"

response = requests.get(url)

soup = BeautifulSoup(response.content, "html.parser")

staff = set()

# Download the stop words corpus

nltk.download('stopwords')

nltk.download('punkt')

def remove(text):

stop\_words = set(stopwords.words('english'))

word\_tokens = word\_tokenize(text.lower())

filtered\_text = [word for word in word\_tokens if word not in stop\_words]

return ' '.join(filtered\_text)

for publication in soup.select(".result-container"):

title = publication.select\_one(".title span").text

authors = [author.text for author in publication.select(".link.person span")]

publisher = publication.select\_one(".link[rel='Publisher'] span")

publication\_status = publication.select\_one(".date").text

publication\_link = publication.select\_one(".title a")["href"]

# Remove stop words from title and authors

title\_without\_stop\_words = remove(title)

authors\_without\_stop\_words = [remove(author) for author in authors]

for author in authors\_without\_stop\_words:

staff.add(author)

(f"Title: {title\_without\_stop\_words}")

(f"Authors: {', '.join(authors\_without\_stop\_words)}")

(f"Publisher: {publisher}")

(f"Publication status: {publication\_status}")

(f"publication Link: {publication\_link}")

(f"CGLauthor: https://pureportal.coventry.ac.uk/en/persons/{author.lower().replace(' ', '-')}")

(f"Number of staff whose publications are crawled: {len(staff)}")

print(f"Title: {title\_without\_stop\_words}")

print(f"Authors: {', '.join(authors\_without\_stop\_words)}")

1.6. When the crawler operates, e.g. scheduled once a week at midnight

import schedule

import time

def automation\_task():

print('Automation task is running...')

# Schedule the automation task to run every 5 seconds

schedule.every(7).days.at('12:00').do(automation\_task)

# You can also schedule the task to run at specific times, for example:

# schedule.every().day.at("10:00").do(automation\_task)

while True:

schedule.run\_pending()

time.sleep(1)

1.5**. Brief explanation of how it works**

The Python code that is offered is a web scraping application that seeks to retrieve information about research publications from a website with a particular URL structure. Explanation base on line of code are as follow:

A.BeautifulSoup and URL: The url variable, which contains the base URL of the website from which we wish to crawl the web page, is first defined in the code. After that, it makes a BeautifulSoup object to parse the response's HTML content and sends a GET request to the URL.

B. Function of research publications: Multiple pages of research publications can be scanned using the research\_publication function. The base URL is a parameter for the function. To keep the retrieved data, it creates a list called publication from scratch.

C. Looping and pagination: Inside While True loop in the research\_publication function runs indefinitely as long as there is new data to crawl. The loop creates the base URLs for the various publishing section page by adding the page number. After that, the function get\_publications\_data is invoked in order to retrieve the data from the url page.

1. get\_publications\_data Function: The get\_publications\_data method (which isn't shown in the code excerpt) is likely a customised function that accepts a URL and returns the pertinent information from that page. The title, authors, publisher, publishing status, and publication link are just a few examples of the specific aspects comprising the publication facts that might be extracted using the BeautifulSoup object.
2. Extraction of Publication Data: A loop over the elements with the publications' data is used to extract the data for each publication.the BeautifulSoup object's "result-container" class. Using the select and select\_one methods, the title, authors, publisher, publication status, and publication link for each publication are extracted.
3. Tracking of staff and author data: To maintain track of the staff members whose publications have been crawled, the algorithm tracks distinct authors in the staff collection.
4. Information on printing publications: The code prints the extracted information for each publication, including the title, authors, publisher, status of the publication, and publishing link. Additionally, based on each author's name, a link to their own page (CGLauthor) is generated.
5. Loop Closure and Courtesy: When there is no more data to extract (publisheddata is empty), the while loop keeps crawling. The code contains a time.sleep(1) command to add a one-second pause between each page request in order to be courteous when scraping. By adding a delay, the possibility of overtaxing the website's servers and being blacklisted for intrusive crawling is diminished.
6. Number of Publications by an Author and Their Maximum Number:

The programme creates an empty dictionary staff to keep track of each employee's number of publications.

Publication Loop and Staff Tracking: The code loops through all of the writers connected to each publication. It verifies that each author is listed in the staff dictionary. If not, it initialises the author's publication count by adding the author as a new key with a value of 0. Then it adds 1 to that author's total number of publications. It also maintains track of the total number of publications for all writers.

Genarally, the code crawls the specified website using BeautifulSoup to gather information about research publications. It shows how to move between a number of pages, extract particular data from the HTML code, and manage data extraction for each publication. The code is made to collect publication data from the website's publications area and determine the most publications per staff member. The information could also be used for reporting, analysis, or any other task needing knowledge of the published data.The code outputs the most publications allowed per employee. The code effectively keeps track of the total number of research publications crawled from the supplied URL and counts the unique staff members. Without providing specific details for each publication, it gives a brief summary of the writers and their works.

**2. Indexer**

2.1. I implemented the index but first passed all the published information to a document:

import requests

from bs4 import BeautifulSoup

url = "https://pureportal.coventry.ac.uk/en/publications/"

response = requests.get(url)

soup = BeautifulSoup(response.content, "html.parser")

publications = []

staff = set()

for publication in soup.select(".result-container"):

title = publication.select\_one(".title span").text

authors = [author.text for author in publication.select(".link.person span")]

publisher = publication.select\_one(".link[rel='Publisher'] span")

publication\_status = publication.select\_one(".date").text

publication\_link = publication.select\_one(".title a")["href"]

for author in authors:

staff.add(author)

doc = [

(f"Title: {title\_without\_stop\_words}"),

(f"Authors: {', '.join(authors\_without\_stop\_words)}"),

(f"Publisher: {publisher}"),

(f"Publication status: {publication\_status}"),

(f"publication Link: {publication\_link}"),

(f"CGLauthor: https://pureportal.coventry.ac.uk/en/persons/{author.lower().replace(' ', '-')}"),

(f"Number of staff whose publications are crawled: {len(staff)}")

]

publications.append(doc)

print(publications)

Initialize an empty inverted index to pass the publication data in the document to it by calling the function in term and posting.

inverted\_index = {}

# Build the inverted index

for document\_id, doc in enumerate(publications):

terms = doc

for term in terms:

if term not in inverted\_index:

inverted\_index[term] = []

inverted\_index[term].append(document\_id)

# Print the inverted index

for term, postings in inverted\_index.items():

print(f"{term}: {postings}")

2.2. it is constructed from scratch every time my crawler is run

2.3. If you implemented it, show some part of its content (e.g. the constructed dictionary)

doc = [

(f"Title: {title\_without\_stop\_words}"),

(f"Authors: {', '.join(authors\_without\_stop\_words)}"),

(f"Publisher: {publisher}"),

(f"Publication status: {publication\_status}"),

(f"publication Link: {publication\_link}"),

(f"CGLauthor: https://pureportal.coventry.ac.uk/en/persons/{author.lower().replace(' ', '-')}"),

(f"Number of staff whose publications are crawled: {len(staff)}")

]

publications.append(doc)

print(publications)

import PySimpleGUI as sg

layout = [[sg.Text('Enter your name:'), sg.InputText()], [sg.Button('Ok'), sg.Button('Cancel')]]

window = sg.Window('Window Title', layout)

while True:

event, values = window.read()

if event == sg.WIN\_CLOSED or event == 'Cancel':

break

print(f'Blechley park {values[0]}!')

window.close()

This code creates a simple window with a text box and two buttons. When the user enters their name and clicks the “Ok” button, the program prints “Blechely park” followed by the user’s name. Overall, the code creates a straightforward interactive window where users can enter their names and, after clicking "Ok," hear a kind welcome. It uses PySimpleGUI, a user-friendly package for easily designing graphical interfaces, to show the fundamentals of creating GUIs.

import schedule

import time

def automation\_task():

print('Automation task is running...')

# Schedule the automation task to run every 7 days at 12am

schedule.every(7).days.at('12:00am').do(automation\_task)

while True:

schedule.run\_pending()

time.sleep(1)

The automation\_task function can be customised to carry out any unique automation or to plan numerous actions at various frequencies or times. It can be manually interrupted by Python programme at the terminal by using Ctrl+C,

2.5. both option A and and task 1 were done in this project

**Option A:**

Essay on Inverted Index:

words or terms is stored in an inverted index, which is a type of data structure. Search engines, database systems, and other applications that require effective text search frequently use inverted indexes.

What an inverted index does is as follows:

Tokenizing, or dissecting materials into individual words or concepts, is the initial stage.

The next step is to establish a mapping from each word to the documents that include that phrase after the papers have been tokenized.

A two-dimensional table is commonly used to record the mapping, with one column for the terms and the other for a list of the documents that each term appears in.

For instance, The inverted index would look like this if we had a document with the terms all, bright, beautiful, and, made, from, God. Entries

|  |  |
| --- | --- |
| term | documents |
| all | 1 |
| bright | 0 |
| beautiful | 1 |
| and | 1 |
| made | 1 |
| from | 0 |
| God | 1 |

For effective text search, inverted indexes are a potent tool. They make it possible for search engines to easily locate all of the publications that include a specific term or phrase. You can do this by simply searching for the word or phrase in the inverted index, after which you can get a list of the publications that are related to it.

There are several benefits to using inverted indexes, including:

They make text searching effective.

They can be used to handle more complicated inquiries, like those that contain several words or sentences.The quality of search results may be improved by using them to rate documents. Inverted indexes can be vast and complex, which is one of its drawbacks, It could be challenging to update them,They could be ineffective for searches including uncommon words or phrases. Inverted indexes are a potent tool with a wide range of applications, despite these drawbacks. The Inverted Index is a fundamental data structure used in information retrieval systems, including search engines like Google Scholar. It provides an efficient way to store and retrieve information based on the terms or keywords present in a document collection. In simple terms, an inverted index maps terms to the documents that contain those terms.

**The Inverted Index consists of two main components:** the dictionary and the postings list. The dictionary stores unique terms or keywords extracted from the documents, along with pointers to their corresponding postings lists. The postings list, for each term, contains references to the documents where the term appears.

**Why is the Inverted Index important? Here are five key reasons:**

1. Fast Document Retrieval: The Inverted Index allows for fast retrieval of documents containing specific terms. Instead of scanning every document in the collection, the search engine can quickly identify relevant documents by looking up the terms in the index and retrieving the associated postings lists.
2. Efficient Space Utilization: The Inverted Index reduces storage space requirements. Instead of storing the entire document text, it only stores the necessary information to identify the documents. By storing the document references in the postings list, the index saves space compared to duplicating the entire document content.
3. Query Optimization: The Inverted Index enables efficient query processing. When a user submits a query, the search engine can quickly identify the relevant terms and retrieve their postings lists. It can then perform operations like intersection or union of postings lists to find documents that satisfy the query conditions.
4. Ranking and Scoring: In addition to document retrieval, the Inverted Index is crucial for ranking and scoring documents. By storing additional information in the postings list, such as term frequency and document weights, the search engine can assign relevance scores to documents and present the most relevant results to users.
5. Scalability: The Inverted Index allows search engines to handle large document collections efficiently. As the number of documents and terms increases, the index can be divided into smaller inverted indexes or distributed across multiple machines, enabling parallel processing and scalable retrieval.

**Here's a simple example of building an inverted index in Python:**

# Sample document collection

documents = [

"This is all the document",

"This document is the last document",

"And this is the second one",

"Is this the first and last document?",

]

# Initialize an empty inverted index

inverted\_index = {}

# Build the inverted index

for doc\_id, document in enumerate(documents):

terms = document.lower().split()

for term in terms:

if term not in inverted\_index:

inverted\_index[term] = []

inverted\_index[term].append(doc\_id)

# Print the inverted index

for term, postings in inverted\_index.items():

print(f"{term}: {postings}")

In the above illustration, we have a straightforward document collection of four papers. We create the inverted index after tokenizing the documents into terms. The list of document IDs where each term appears is mapped to the list. The posts lists for each phrase are displayed when printing the inverted index. Baeldung (2022)

3.1 Which pre-processing tasks are applied to a given query

Query processor

Query processor these actions are designed to standardise and improve the query's meaning in order to make it easier to retrieve accurate and pertinent information.

#query preprocessing

def get\_publications\_data(url):

response = requests.get(url)

if response.status\_code == 200:

return BeautifulSoup(response.content, "html.parser")

return None

def process\_publications(url):

staff = {}

publications = []

def preprocess\_query(query):

soup = get\_publications\_data(url)

# Tokenization

tokens = word\_tokenize(query)

# Lowercasing

tokens = [token.lower() for token in tokens]

# Remove special characters and non-alphanumeric characters

tokens = [re.sub(r'[^a-zA-Z0-9]', '', token) for token in tokens]

# Stop word removal

stop\_words = set(stopwords.words('english'))

tokens = [token for token in tokens if token not in stop\_words]

# Stemming

stemmer = PorterStemmer()

tokens = [stemmer.stem(token) for token in tokens]

# Join the tokens back into a cleaned query

cleaned\_query = ' '.join(tokens)

for publication in soup.select(".result-container"):

title = publication.select\_one(".title span").text

authors = [author.text for author in publication.select(".link.person span")]

publisher = publication.select\_one(".link[rel='Publisher'] span")

publication\_status = publication.select\_one(".date").text.strip()

link = publication.select\_one(".title a")["href"]

for author in authors:

if author not in staff:

staff[author] = 0

staff[author] += 1

dict = {

"Title": title,

"Authors": authors,

"Publisher": publisher,

"Publication Status": publication\_status,

"Publication Link": link

}

publications.append(dictionary)

return publications, cleaned\_query

if \_\_name\_\_ == "\_\_main\_\_":

url = "https://pureportal.coventry.ac.uk/en/publications/"

publications\_data = process\_publications(url)

#print(publications, cleaned\_query)

#print()

print(f"Original Query:", query)

print(f"Cleaned Query:", cleaned\_query)

#print("Maximum Publications Information:")

3.2. Elastic search was not used

3.3.did not perform rank retrieval

3.4. Brief explanation of how it works

The preprocess\_query function processes a query using the following preprocessing operations as input:

Tokenisation of data: The word\_tokenize function from NLTK is used to separate the query into its component terms. Lowercase: Using a list comprehension, every word in the query is changed to lowercase.

Removal of Special Characters and Non-Alphabetic Characters: We remove any special characters and non-alphanumeric characters from each word using regular expressions (re).

Stop Word Removal using the NLTK's English stop words list, we eliminate frequent stop words (such as "the," "a," and "is"), Stemming to break each word down to its root or base form, we use the Porter Stemmer from NLTK.

**4. (Optional)**

Any other important point you may want to mention, including any restriction, extras, issues:

**Note**; that the provided code is only a fragment and is deficient in several areas, including adequate error handling, database storage, and writing the data to a file for additional analysis. For a reliable and scalable web scraping solution in the real world, extra error handling and data storage mechanisms should be taken into account.

**Part 2 – Text classification**

1. How training data were collected

import requests

from bs4 import BeautifulSoup

import re

import random

from sklearn.feature\_extraction.text import CountVectorizer

from sklearn.naive\_bayes import MultinomialNB

from sklearn.metrics import accuracy\_score, classification\_report

from sklearn.model\_selection import train\_test\_split

import pandas as pd

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.naive\_bayes import MultinomialNB

import nltk

from nltk.corpus import stopwords

from nltk.tokenize import word\_tokenize

from nltk.stem import PorterStemmer

from sklearn.metrics import accuracy\_score, classification\_report

import pandas as pd

df = pd.read\_csv('bbc-text.csv')

df.head()

# Summary statistics

#print("\nSummary statistics:")

df.describe()

df.info()

#to check class imbalance

df['text'].value\_counts()

class TextClassifier:

def \_\_init\_\_(self):

self.tfidf\_vectorizer = TfidfVectorizer()

self.text\_classifier = MultinomialNB()

def preprocess\_text(self, text):

# Convert text to lowercase

text = text.lower()

# Tokenize the text

tokens = word\_tokenize(text)

# Remove punctuation and numbers

tokens = [word for word in tokens if word.isalpha()]

# Remove stopwords

stop\_words = set(stopwords.words('english'))

tokens = [word for word in tokens if word not in stop\_words]

# Stemming

stemmer = PorterStemmer()

tokens = [stemmer.stem(word) for word in tokens]

# Combine tokens back to text

processed\_text = ' '.join(tokens)

return processed\_text

def train(self, X\_train, y\_train):

processed\_X\_train = X\_train.apply(self.preprocess\_text)

X\_train\_tfidf = self.tfidf\_vectorizer.fit\_transform(processed\_X\_train)

self.text\_classifier.fit(X\_train\_tfidf, y\_train)

def classify\_text(self, text):

processed\_text = self.preprocess\_text(text)

tfidf\_text = self.tfidf\_vectorizer.transform([processed\_text])

class\_label = self.text\_classifier.predict(tfidf\_text)[0]

return class\_label

df = pd.read\_csv('bbc-text.csv')

# Create an instance of TextClassifier

text\_classifier = TextClassifier()

# Train the classifier

text\_classifier.train(df['text'], df['category'])

sentence = " he asked mr myers if he ever knew mr ebbers make an accounting decision "

predicted\_classes = text\_classifier.classify\_text(sentence)

print("Predicted Classes:", predicted\_classes)

sentences = {

'text': [

'tv future in the hands of viewers with home theatre systems.',

'The has firmly declared his innocence.',

'so far this year they have compared me to fagin to shylock and to a flying pig.'

],

'category': ['tech', 'Sport', 'business']

}

df = pd.read\_csv('bbc-text.csv')

X\_train, X\_test, y\_train, y\_test = train\_test\_split(df['text'], df['category'], test\_size=0.2, random\_state=42)

vectorizer = TfidfVectorizer()

X\_train\_vectorized = vectorizer.fit\_transform(X\_train)

X\_test\_vectorized = vectorizer.transform(X\_test)

classifier = MultinomialNB()

classifier.fit(X\_train\_vectorized, y\_train)

y\_pred = classifier.predict(X\_test\_vectorized)

print("Accuracy:", accuracy\_score(y\_test, y\_pred))

print("\nClassification Report:")

print(classification\_report(y\_test, y\_pred))

sentences is initially defined as a dictionary before being transformed into a pandas DataFrame. Using train\_test\_split from the scikit-learn library, we then divided the data into training and testing sets. The text input is subsequently transformed into numerical vectors using a TF-IDF vectorizer. The relevance of each word in each document with relation to the overall dataset is represented by the TF-IDF vectors. Developed a Multinomial Naive Bayes classifier after vectorizing the data, which is appropriate for text classification applications. Using the fit method, we train the classifier on the training set of data. The predict technique is then used with the trained classifier to make predictions on the test set. Finally, we calculate accuracy and publish a classification report to assess the model's performance.

2. Which classification method has been used and how its performance is measured

Multiclass classification method was used.

3. Brief explanation of how it works:

We import the necessary libraries such as pandas, scikit-learn, and nltk.

We define a class called TextClassifier which has three methods:

\_\_init\_\_: Initializes the TfidfVectorizer and MultinomialNB objects.

preprocess\_text: Preprocesses the text by converting it to lowercase, tokenizing it, removing punctuation and numbers, removing stopwords, stemming the words, and combining the words back to text.

train: Trains the classifier on the input data.

classify\_text: Classifies the input text into one or more categories.

Read in the data from a CSV file using pandas. Created an instance of the TextClassifier class, trained train the classifier on the input data.

classified a sentence using the classifier and print out the predicted classe statement. sentences is initially defined as a dictionary before being transformed into a pandas DataFrame. Using train\_test\_split from the scikit-learn library, we then divided the data into training and testing sets. The text input is subsequently transformed into numerical vectors using a TF-IDF vectorizer. The relevance of each word in each document with relation to the overall dataset is represented by the TF-IDF vectors. Developed a Multinomial Naive Bayes classifier after vectorizing the data, which is appropriate for text classification applications. Using the fit method, we train the classifier on the training set of data. The predict technique is then used with the trained classifier to make predictions on the test set. Finally, we calculate accuracy and publish a classification report to assess the model's performance.

1. I implemented the text classification code and also did option B

**TASK 2 OPTION B**

Giving a specific piece of text a name or category is the process of text classification. Although it can be carried out manually, machine learning techniques are frequently used to carry it out automatically. Natural language processing (NLP) tasks like text classification are ubiquitous and have many uses.Manning et al. (2008)

Spam filtering is a typical use for text classification. Emails that are most likely to be spam are identified by spam filters using text categorization. This is accomplished by scanning the email's text for terms or phrases that are frequently connected to spam. For instance, a message with the phrases "free" and "money" is more likely to be spam than one with the words "business" and "proposal."

another typical.Sentiment analysis is a text categorization application. The method of figuring out a text's emotional tone is called sentiment analysis. This can be achieved by scrutinising the language used in the text and the context in which it is used. For instance, an email with the subject line "I love your product" is more likely to receive a positive response than one with the subject line "I'm not happy with your service."

Documents, topics, and keywords can all be extracted using text classification in addition to document categorization. A text classifier might be used, for instance, to extract the names of people and organisations from a legal document or to identify the essential terms in a product description.

The two fundamental techniques for categorising texts are supervised learning and unsupervised learning McCallum & Nigam (1998). Using supervised learning, a set of labelled data is used to train the text classifier. This data consists of text documents that have been previously categorised into one or more categories. After training to distinguish the features associated with each category, the text classifier uses these attributes to categorise fresh text documents. The text classifier is not trained on labelled data in unsupervised learning. Instead, a series of unlabeled text documents is sent to the text classifier, and it learns to recognise the categories to which these documents belong. This is accomplished by grouping the documents according to how similar they are.

With the use of text classification, one can effectively mine text data for insightful information Rajan & Radev (2019). It has a wide range of uses, and as text data continues to increase in volume, its significance is rising.

The following are a few advantages of text classification:

It can aid in the organisation and structuring of text data, the discovery of patterns and trends in text data, the extraction of keywords and other significant information from text data, and the automation of processes that would otherwise be carried out manually.

The following are some difficulties with text classification:

The accuracy of the text classifier can be significantly impacted by the calibre of the text data.

If the dataset used for training the text classifier is not representative of the population of text documents that it will be used to classify, the results of the classifier may be biassed.

The text classifier's deployment and training can be computationally expensive.

Despite these difficulties, text categorization is an effective tool that may be employed to draw insightful conclusions from text data. Text classification is going to become progressively more crucial as the volume of text data increases.

**What is text classification, first off?**

Text categorization, sometimes referred to as text categorization or document categorization, is a fundamental problem in natural language processing (NLP) that entails automatically labelling or categorising texts. The objective is to teach a machine learning model to identify patterns and features in textual data so that it can accurately categorise new, unknown material.

2. Major Text Classification Types: The following categories can be used to classify text classification broadly:

A binary classification divides the text into two categories that are mutually exclusive. One well-known instance is spam email detection, which involves determining whether or not an email is spam.

a. Multiclass Classification: Text in this type is categorised into a single class.consisting of several non-exclusive groups. For instance, there are numerous categories into which news stories can be divided, including sports, politics, entertainment, and technology.

c. Multilabel Classification: This approach involves giving a single text document numerous labels. Positive, negative, and neutral sentiment tags may be used to indicate various facets of a stated sentiment in a product evaluation.

3. Text categorization has usage in many different fields, and among of its most significant use cases are as follows:

Spam detection uses binary classification techniques to distinguish between spam and valid emails.vast amounts of textual information.

Language detection, which is useful for multilingual NLP applications, identifies the language in which a given text was authored.

e. Automating the classification of documents in digital libraries or news stories into appropriate groups.

4. Text Classification Using Naive Bayes: Simple Python Example

For the purpose of classifying textual data into predetermined groups, text classification is an essential NLP activity. Binary classification, multiclass classification, and multilabel classification are some of its most common types. Applications for text classification include document categorization, language detection, topic modelling, sentiment analysis, and spam detection. Simple methods for implementing text classification models, like the Naive Bayes example shown above, are provided by Python's scikit-learn module.

REFERENCE

Manning, C. D., Raghavan, P., & Schütze, H. (2008). Introduction to information retrieval. Cambridge, UK: Cambridge University Press.

McCallum, A. K., & Nigam, K. (1998). A comparison of event models for naive Bayes text classification. In AAAI Workshop on Learning for Text Categorization (pp. 41-48).

Rajan, D., & Radev, D. (2019). Text classification: Theory and applications. Cambridge, UK: Cambridge University Press.

Baeldung. (2022, January 25). Indexing: Inverted Index. Baeldung. Retrieved from https://www.baeldung.com/cs/indexing-inverted-index

Microsoft bing search:Scikit-learn documentation: [https://scikit-learn.org/stable/documentation.html](https://scikit-learn.org/stable/documentation.html" \t "https://chat.openai.com/c/_new)

"Natural Language Processing in Action" by Lane, Howard, and Hapke (Manning Publications)

**Appendix Screenshoot**































































