**Faculty of Engineering, Environment and Computing (EEC 7071CEM)**

**Information Retrieval**

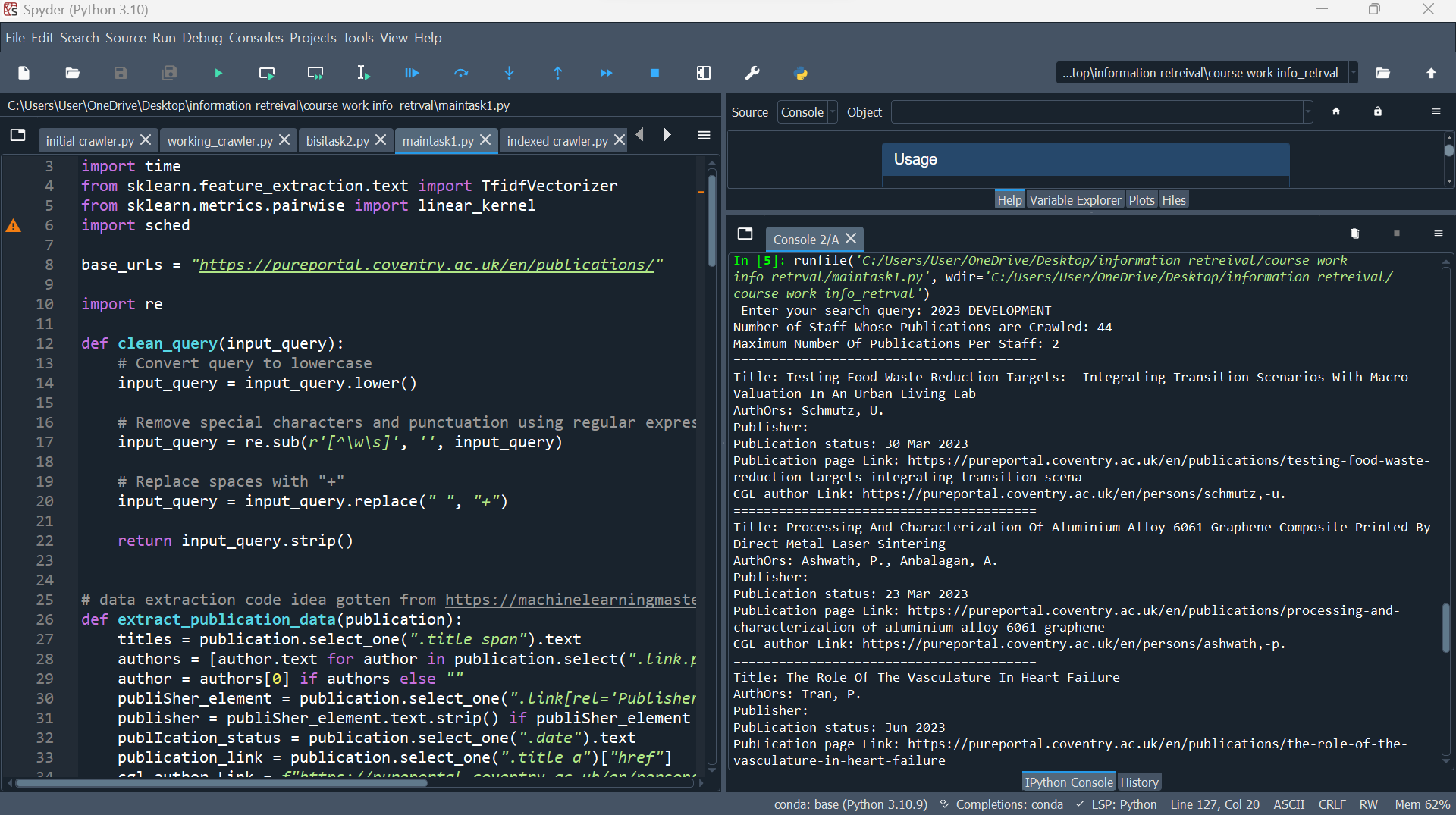
**Obisanya Babatunde Adedeji (13311980)**

**TASK 1**

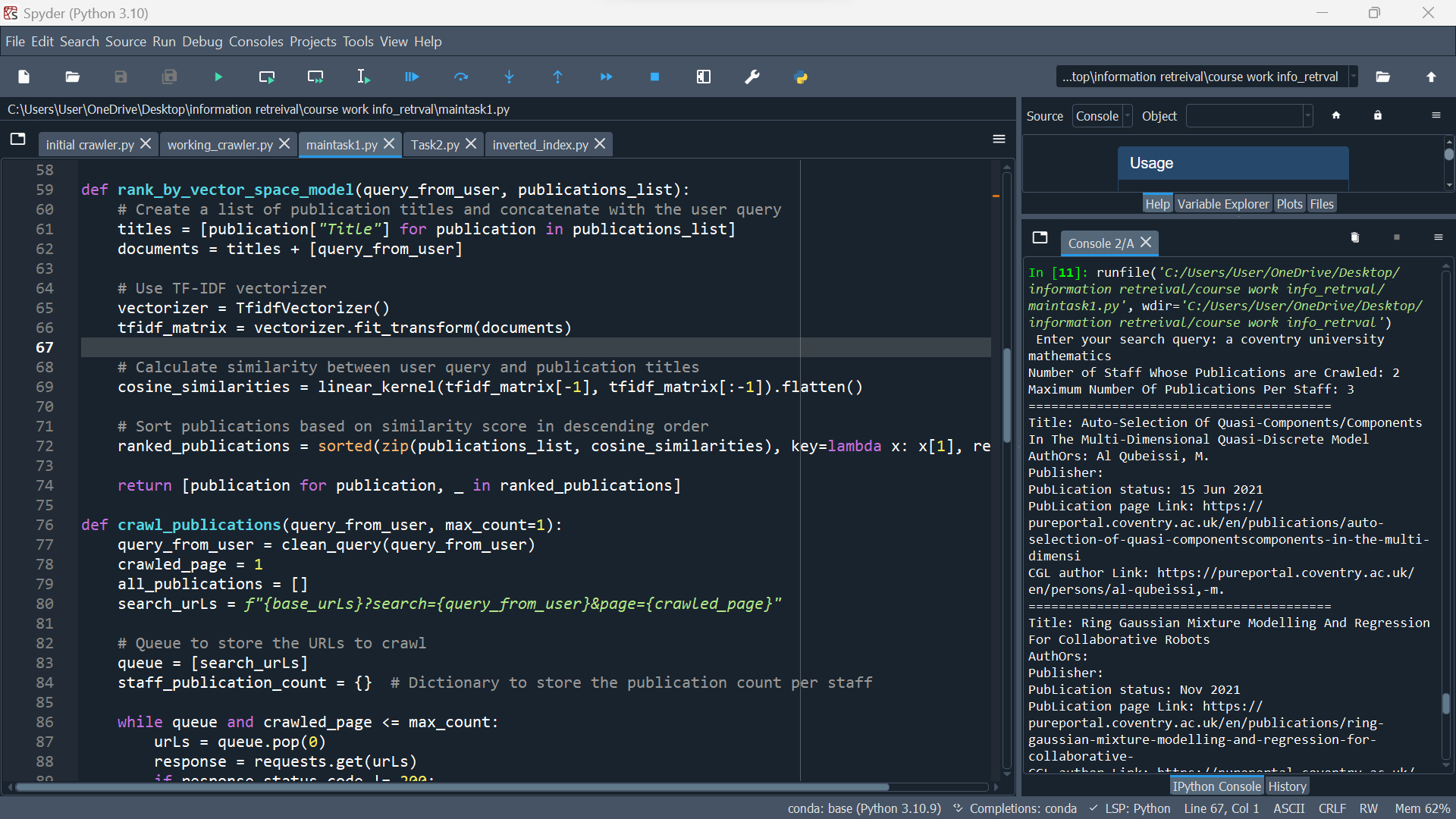
The code takes a user's search query as input. It crawls the web portal using the query, extracts publication data, and stocks it in a list of dictionaries. The crawling process repeats for multiple pages of search results. The crawled data is pre-processed (capitalization, lowercase conversion) and then ranked using the Vector Space Model (TF-IDF) based on the user's query to retrieve the most relevant publications. The top 10 ranked publications are displayed.

The code maintains a dictionary called staff\_publication\_count to keep track of the number of publications made by each staff member. It raises the count for each publication connected to a writer (staff member) during the crawling procedure. The number of staff members whose publications are crawled can be calculated using the length of this dictionary (len(staff\_publication\_count)). The information gotten about each publication includes Title, Authors (multiple authors are concatenated with commas), Publisher, Publication status, Publication page link, CGL author link (URL to the author's profile).

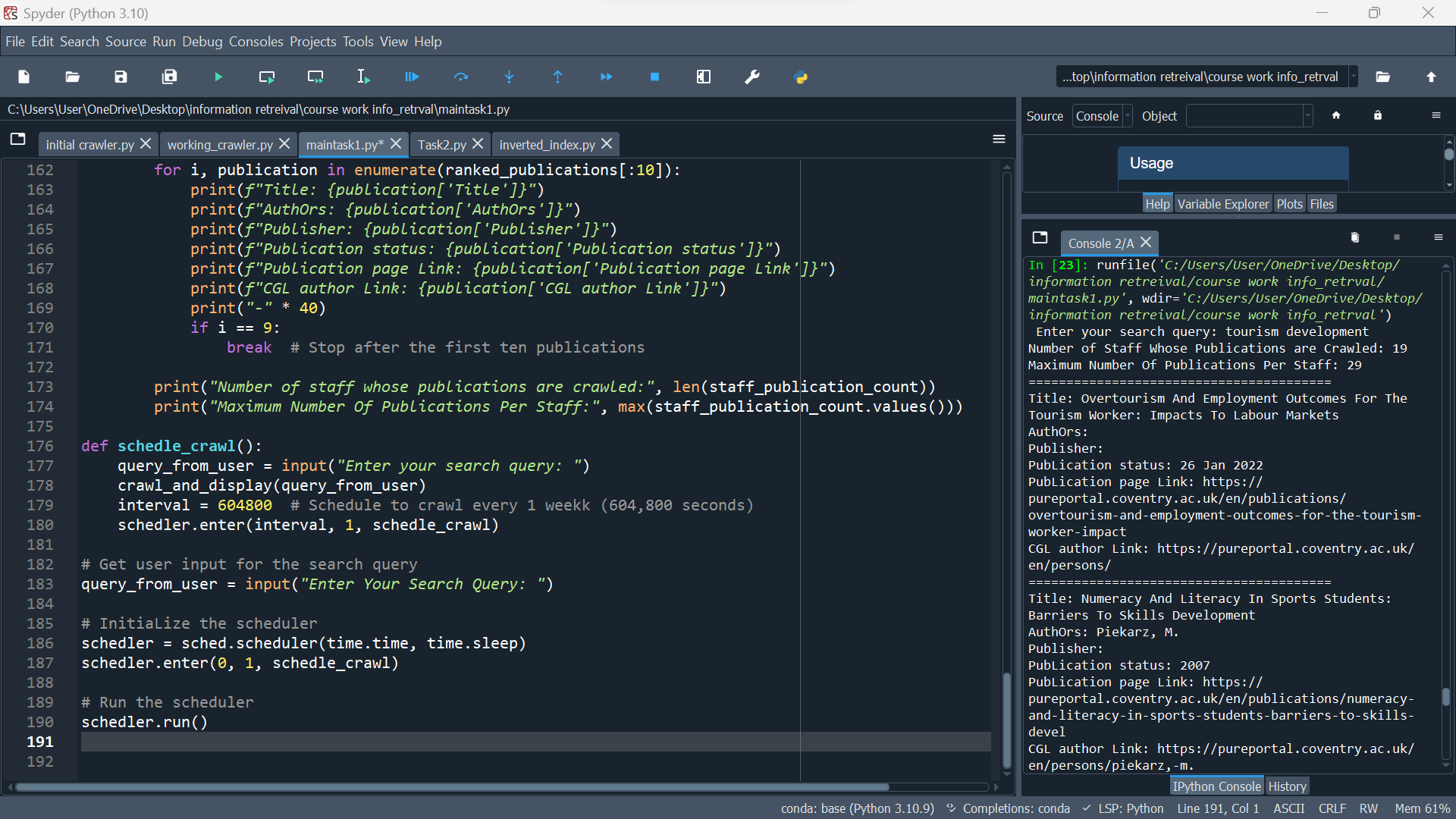
The pre-processing tasks performed on output are making the first letter of each word in the title capital, Removing leading/trailing whitespaces from author names. Changing publisher names to lowercase. The code runs the crawler and display the results. Also, it has a schedle\_crawl() function that uses a scheduler to crawl the publications periodically (every one week or 604,800 seconds).



**Using both integer and string with space as sample user query:** 2023 DEVELOPMENT



**Using stop word, string with space as sample user query:** a coventry university mathematics



**Using two strings with space as sample user query:** tourism development

**import** **requests**

**from** **bs4** **import** BeautifulSoup

**import** **time**

**from** **sklearn.feature\_extraction.text** **import** TfidfVectorizer

**from** **sklearn.metrics.pairwise** **import** linear\_kernel

**import** **sched**

base\_urLs = "https://pureportal.coventry.ac.uk/en/publications/"

**import** **re**

**def** **clean\_query**(input\_query):

# Convert query to lowercase

input\_query = input\_query.lower()

# Remove special characters and punctuation using regular expression

input\_query = re.sub(r'[^\w\s]', '', input\_query)

# Replace spaces with "+"

input\_query = input\_query.replace(" ", "+")

**return** input\_query.strip()

# data extraction code idea gotten from https://machinelearningmastery.com/web-crawling-in-python/

**def** **extract\_publication\_data**(publication):

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

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

author = authors[**0**] **if** authors **else** ""

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

publisher = publiSher\_element.text.strip() **if** publiSher\_element **else** ""

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

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

cgl\_author\_Link = f"https://pureportal.coventry.ac.uk/en/persons/{author.lower().replace(' ', '-')}"

**return** {

"Title": titles,

"AuthOrs": ', '.join(authors),

"Publisher": publisher,

"Publication status": publIcation\_status,

"Publication page Link": publication\_link,

"CGL author Link": cgl\_author\_Link

}

**def** **preprocess\_publication\_data**(publications\_list):

**for** publication **in** publications\_list:

# Capitalize the first letter of each word in the title

publication["Title"] = publication["Title"].title()

# Remove leading and trailing whitespaces from author names

authors = [author.strip() **for** author **in** publication["AuthOrs"].split(",")]

publication["AuthOrs"] = ", ".join(authors)

# Convert publisher names to lowercase

publication["Publisher"] = publication["Publisher"].lower()

**return** publications\_list

**def** **rank\_by\_vector\_space\_model**(query\_from\_user, publications\_list):

# Create a list of publication titles and concatenate with the user query

titles = [publication["Title"] **for** publication **in** publications\_list]

documents = titles + [query\_from\_user]

# Use TF-IDF vectorizer

vectorizer = TfidfVectorizer()

tfidf\_matrix = vectorizer.fit\_transform(documents)

# Calculate similarity between user query and publication titles

cosine\_similarities = linear\_kernel(tfidf\_matrix[-**1**], tfidf\_matrix[:-**1**]).flatten()

# Sort publications based on similarity score in descending order

ranked\_publications = sorted(zip(publications\_list, cosine\_similarities), key=**lambda** x: x[**1**], reverse=True)

**return** [publication **for** publication, \_ **in** ranked\_publications]

**def** **crawl\_publications**(query\_from\_user, max\_count=**1**):

query\_from\_user = clean\_query(query\_from\_user)

crawled\_page = **1**

all\_publications = []

search\_urLs = f"{base\_urLs}?search={query\_from\_user}&page={crawled\_page}"

# Queue to store the URLs to crawl

queue = [search\_urLs]

staff\_publication\_count = {} # Dictionary to store the publication count per staff

**while** queue **and** crawled\_page <= max\_count:

urLs = queue.pop(**0**)

response = requests.get(urLs)

**if** response.status\_code != **200**:

**print**("Failed to retrieve data. Please try again later.")

**return** all\_publications

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

publications = soup.select(".result-container")

**if** **not** publications:

**break**

**for** publication **in** publications:

publication\_data = extract\_publication\_data(publication)

all\_publications.append(publication\_data)

# Count the number of publications per staff

author = publication\_data['AuthOrs'].split(', ')[**0**]

staff\_publication\_count[author] = staff\_publication\_count.get(author, **0**) + **1**

crawled\_page += **1**

next\_urLs = f"{base\_urLs}?search={query\_from\_user}&page={crawled\_page}"

queue.append(next\_urLs)

time.sleep(**2**) #preserves the robots.txt rules

**return** all\_publications, staff\_publication\_count

# Get user input for the search query

query\_from\_user = input(" Enter your search query: ")

publications\_list, staff\_publication\_count = crawl\_publications(query\_from\_user)

**if** **not** publications\_list:

**print**("No Publications Found.")

**else**:

# Apply preprocessing tasks to the crawled data

publications\_list = preprocess\_publication\_data(publications\_list)

**print**("Number of Staff Whose Publications are Crawled:", len(staff\_publication\_count))

**print**("Maximum Number Of Publications Per Staff:", max(staff\_publication\_count.values()))

**print**("=" \* **40**)

# Rank publications using the Vector Space Model

ranked\_publications = rank\_by\_vector\_space\_model(query\_from\_user, publications\_list)

**for** i, publication **in** enumerate(ranked\_publications[:**10**]):

**print**(f"Title: {publication['Title']}")

**print**(f"AuthOrs: {publication['AuthOrs']}")

**print**(f"Publisher: {publication['Publisher']}")

**print**(f"PubLication status: {publication['Publication status']}")

**print**(f"PubLication page Link: {publication['Publication page Link']}")

**print**(f"CGL author Link: {publication['CGL author Link']}")

**print**("=" \* **40**)

**if** i == **5**:

**break** # Stop after the first ten publications

**print**("Number Of Staff Whose Publications Are Crawled:", len(staff\_publication\_count))

**print**("Maximum Number Of Publications Per Staff:", max(staff\_publication\_count.values()))

**def** **crawl\_and\_display**(query\_from\_user):

publications\_list, staff\_publication\_count = crawl\_publications(query\_from\_user)

**if** **not** publications\_list:

**print**("No publications found.")

**else**:

# Apply preprocessing tasks to the crawled data

publications\_list = preprocess\_publication\_data(publications\_list)

**print**("Number of staff whose publications are crawled:", len(staff\_publication\_count))

**print**("Maximum number of publications per staff:", max(staff\_publication\_count.values()))

# Rank publications using the Vector Space Model

ranked\_publications = rank\_by\_vector\_space\_model(query\_from\_user, publications\_list)

**for** i, publication **in** enumerate(ranked\_publications[:**10**]):

**print**(f"Title: {publication['Title']}")

**print**(f"AuthOrs: {publication['AuthOrs']}")

**print**(f"Publisher: {publication['Publisher']}")

**print**(f"Publication status: {publication['Publication status']}")

**print**(f"Publication page Link: {publication['Publication page Link']}")

**print**(f"CGL author Link: {publication['CGL author Link']}")

**print**("-" \* **40**)

**if** i == **9**:

**break** # Stop after the first ten publications

**print**("Number of staff whose publications are crawled:", len(staff\_publication\_count))

**print**("Maximum Number Of Publications Per Staff:", max(staff\_publication\_count.values()))

**def** **schedle\_crawl**():

query\_from\_user = input("Enter your search query: ")

crawl\_and\_display(query\_from\_user)

interval = **604800** # Schedule to crawl every 1 weekk (604,800 seconds)

schedler.enter(interval, **1**, schedle\_crawl)

# Get user input for the search query

query\_from\_user = input("Enter Your Search Query: ")

# InitiaLize the scheduler

schedler = sched.scheduler(time.time, time.sleep)

schedler.enter(**0**, **1**, schedle\_crawl)

# Run the scheduler

schedler.run()

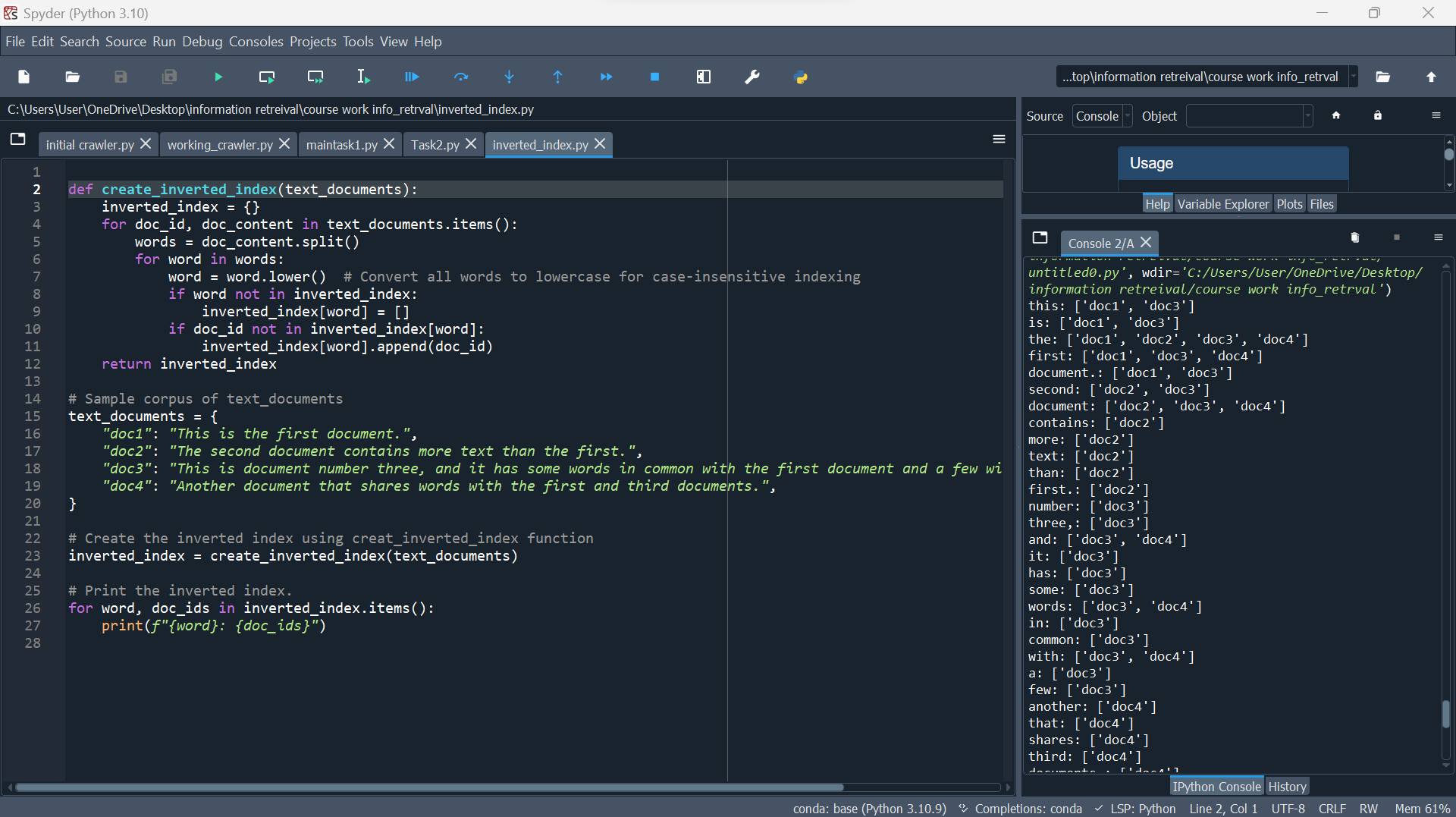
**OPTION A**

An inverted index is a crucial data structure used in information retrieval systems to facilitate quick and efficient search operations. It serves as a map between words and the documents that contain them. In traditional book indexes, we look up words to find the pages they appear. In contrast, an inverted index allows us to find papers based on the words they contain [1]

**Importance of Inverted Index:**

1. **Quick Retrieval**: With an inverted index, searching for documents having specific words becomes highly efficient. Instead of scanning through the entire document group, the index directs us to relevant documents directly, extremely reducing search times.
2. **Reduced Storage:** Inverted indexes are more space-efficient compared to storing full documents. By storing word incidences and document references, the index consumes fewer storage space, making it good for large-scale information retrieval systems.
3. **Support for Ranking:** Inverted indexes lay the foundation when it comes to ranking algorithms like TF-IDF (Term Frequency-Inverse Document Frequency). These procedures facilitate the prioritizing of search results based on word occurrences, enhancing result relevance.

**Example of inverted index**

****

The create\_inverted\_index function takes a dictionary of documents and creates an inverted index, mapping words to the list of document IDs where they appear. It converts words to lowercase for case-insensitive indexing. The function then iterates in the documents, splits the content into words, and then populates the inverted index accordingly. Finally, it returns the index while allowing for efficient word-based document retrieval. The code prints the inverted index to show a word with its corresponding list of document IDs where it appears.

In summary, the code efficiently constructs an inverted index that allows for fast and efficient keyword-based information and document retrieval.

**def** **create\_inverted\_index**(text\_documents):

inverted\_index = {}

**for** doc\_id, doc\_content **in** text\_documents.items():

words = doc\_content.split()

**for** word **in** words:

word = word.lower() # Convert all the words to lowercase for case\_insensitive indexing

**if** word **not** **in** inverted\_index:

inverted\_index[word] = []

**if** doc\_id **not** **in** inverted\_index[word]:

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

**return** inverted\_index

# Sampl text documents

text\_documents = {

"doc1": "This is the first document.",

"doc2": "The second document contains more text than the first.",

"doc3": "This is document number three, and it has some words in common with the first document and a few with second document.",

"doc4": "Another document that shares words with the first and third documents.",

}

# Create the inverted index using function creat\_inverted\_index

inverted\_index = create\_inverted\_index(text\_documents)

# Print out the inverted index.

**for** word, doc\_ids **in** inverted\_index.items():

**print**(f"{word}: {doc\_ids}")

**TASK 2**

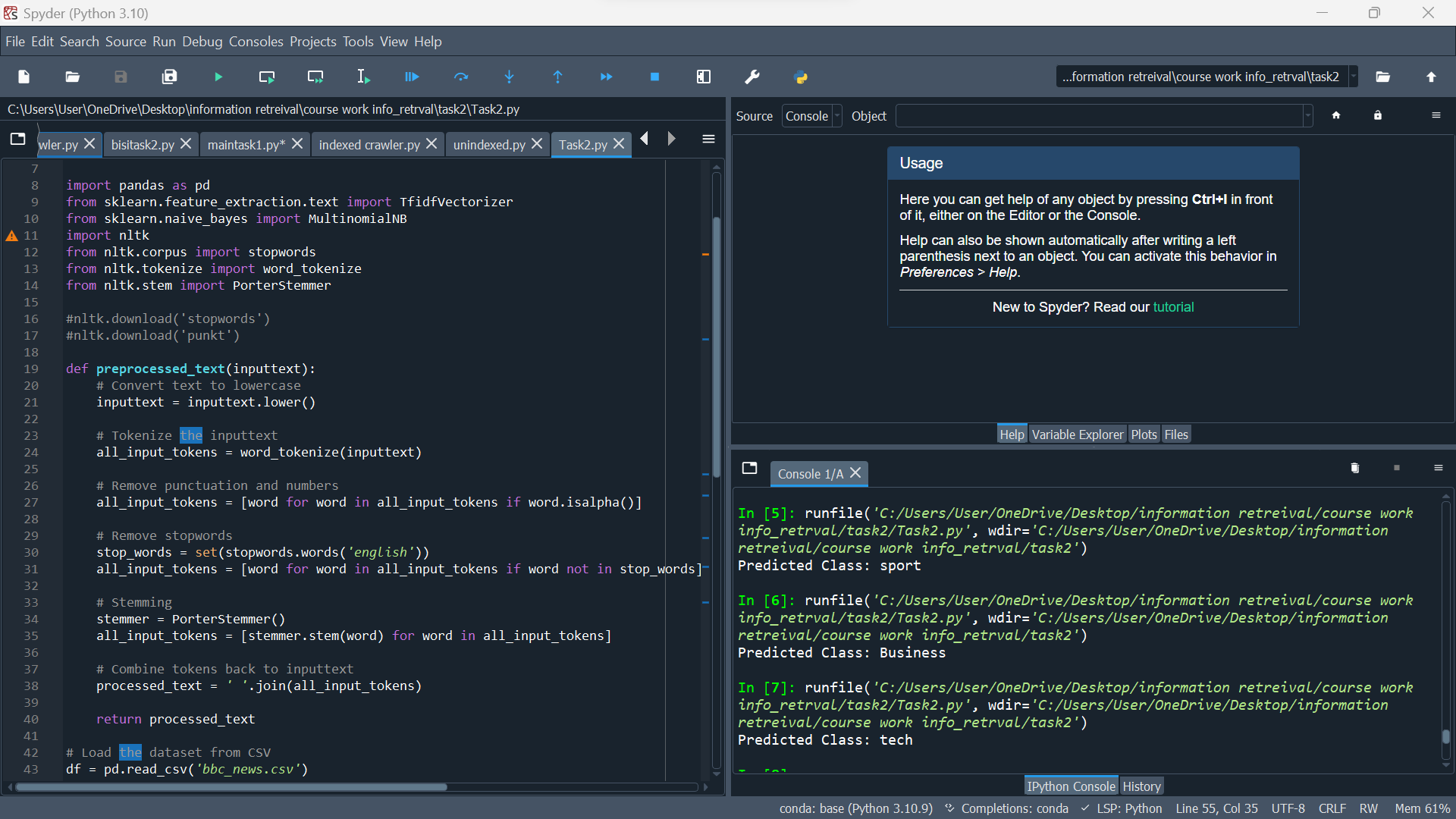
**How Training Data Were Collected**

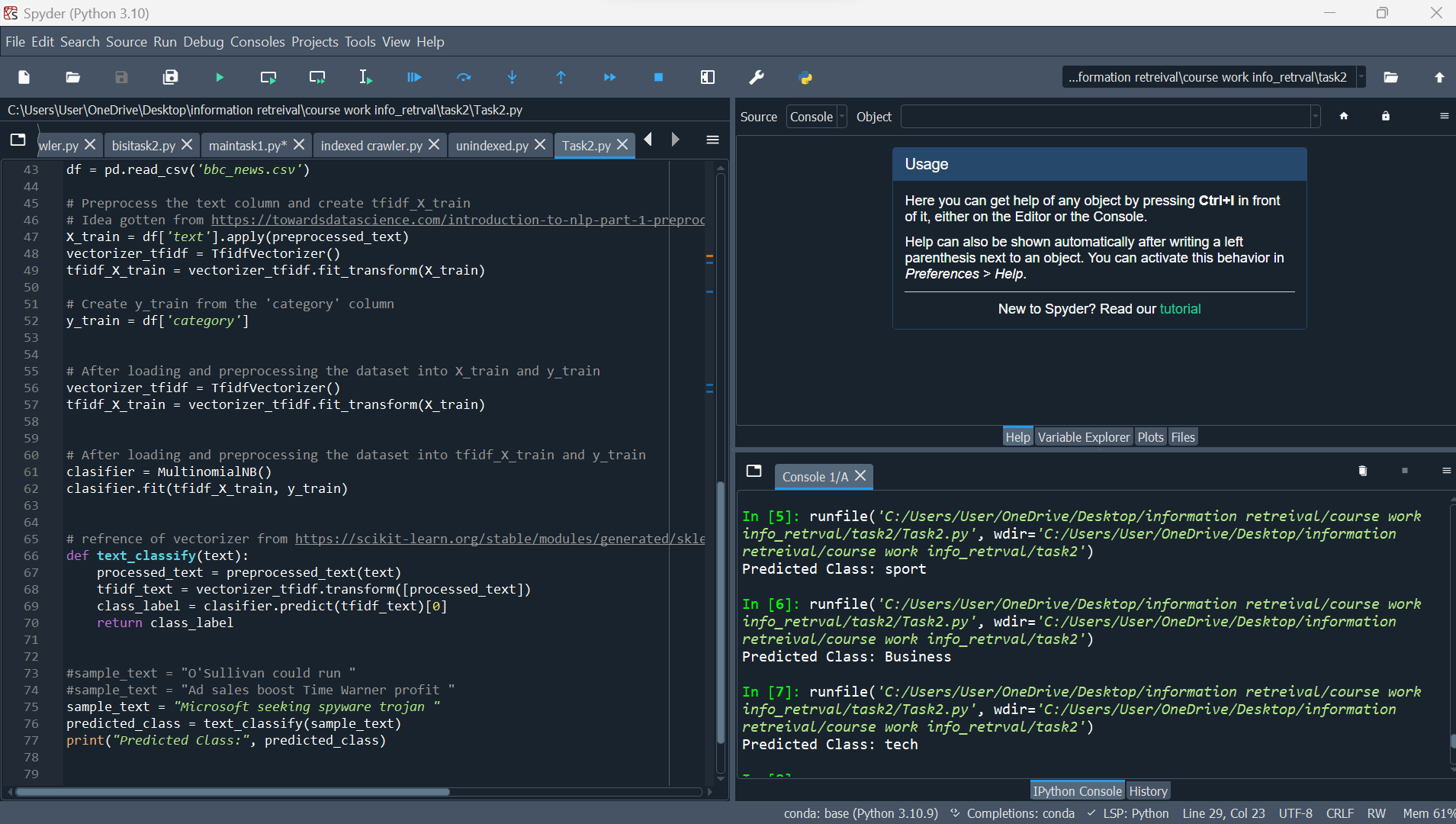
The bbc news dataset from the bbc news website for datasets [2] was used for the classification system which is available in a CSV file named 'bbc\_news.csv'. The dataset is then loaded into a pandas DataFrame using the pd.read\_csv function. The dataset contains labeled news articles with two columns which are 'text' containing the textual content of the news articles and 'category' containing the corresponding categories for each article (business, sport and tech). The code uses the Multinomial Naive Bayes (MNB) classifier for text classification. Naive Bayes is a probabilistic algorithm based on Bayes' theorem and is well-suited for text classification tasks [3]. The MNB classifier assumes that the words are conditionally independent given the class label. Below is a systematic explanation of how the system works:

The libraries required are imported, such as pandas for data handling, scikit-learn for the classifier and vectorization, and nltk for natural language processing tasks. The preprocessed\_text function is defined to preprocess the input text. It changes the text to lowercase, tokenizes it, removes punctuation and numbers, removes stopwords, and applies stemming to decrease words to their root form.

The dataset is loaded from 'bbc\_news.csv', and the 'text' column is preprocessed using the function preprocessed\_text to create the X\_train variable containing preprocessed textual documents. The TfidfVectorizer from scikit-learn is used to translate the text data (X\_train) into a numerical format, where each word is denoted as a numeric value based on its TF-IDF score. The transformed data is stored in tfidf\_X\_train. The target labels are stored in y\_train, which matches to the 'category' column in the CSV file. An instance of the Multinomial Naive Bayes classifier is created and trained on the TF-IDF converted data (tfidf\_X\_train) and target labels (y\_train). The function text\_classify takes an input text, preprocesses it, changes it into TF-IDF representation, and classifies its class label using the trained classifier.

The code then delivers a sample text (sample\_text) for classification, and the function text\_classify predicts its class label, which is printed as "Predicted Class" to the user.





**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

#nltk.download('stopwords')

#nltk.download('punkt')

**def** **preprocessed\_text**(inputtext):

# Convert text to lowercase

inputtext = inputtext.lower()

# Tokenize the inputtext from user

all\_input\_tokens = word\_tokenize(inputtext)

# Remove the punctuation and numbers

all\_input\_tokens = [word **for** word **in** all\_input\_tokens **if** word.isalpha()]

# Remove stopwords

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

all\_input\_tokens = [word **for** word **in** all\_input\_tokens **if** word **not** **in** stop\_words]

# Stem

stemmer = PorterStemmer()

all\_input\_tokens = [stemmer.stem(word) **for** word **in** all\_input\_tokens]

# Combine tokens back into inputtext

processed\_text=' '.join(all\_input\_tokens)

**return** processed\_text

# Load the dataset from CSV

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

# Preprocess the text column and create tfidf\_X\_train

# Idea gotten from https://towardsdatascience.com/introduction-to-nlp-part-1-preprocessing-text-in-python-8f007d44ca96

X\_train = df['text'].apply(preprocessed\_text)

vectorizer\_tfidf = TfidfVectorizer()

tfidf\_X\_train = vectorizer\_tfidf.fit\_transform(X\_train)

# Create y\_train from the 'category' column

y\_train = df['category']

# After loading and preprocessing the dataset into X\_train and y\_train

vectorizer\_tfidf = TfidfVectorizer()

tfidf\_X\_train = vectorizer\_tfidf.fit\_transform(X\_train)

# After loading and preprocessing the dataset into tfidf\_X\_train and y\_train

clasifier = MultinomialNB()

clasifier.fit(tfidf\_X\_train, y\_train)

# refrence of vectorizer from https://scikit-learn.org/stable/modules/generated/sklearn.feature\_extraction.text.TfidfVectorizer.html

**def** **text\_classify**(text):

processed\_text = preprocessed\_text(text)

tfidf\_text = vectorizer\_tfidf.transform([processed\_text])

class\_label = clasifier.predict(tfidf\_text)[**0**]

**return** class\_label

#sample\_text = "O'Sullivan could run "

#sample\_text = "Ad sales boost Time Warner profit "

sample\_text = "Microsoft seeking spyware trojan "

predicted\_class = text\_classify(sample\_text)

**print**("Predicted Class:", predicted\_class)

**REFERENCES**

# [1] A brief explanation of the Inverted Index:

<https://medium.com/@igorkopanev/a-brief-explanation-of-the-inverted-index-f082993f8605>

[2] [Hugging Face](https://huggingface.co/) : <https://huggingface.co/datasets/SetFit/bbc-news>

[3] A brief review of Bayesian statistics:

<https://www.ibm.com/topics/naive-bayes#:~:text=The%20Na%C3%AFve%20Bayes%20classifier%20is,a%20given%20class%20or%20category>.