Task 2: Text Analysis

1. Provide a jupyter notebook that performs the following text analysis tasks on this dataset:

a. Process the news headline text:

i. Parse the JSON files, and extract the text from each record (extract the data in “headline” and “short description”).

In [ ]:

```
import json

# Open the file and read the contents
with open('NewsCategoryDataset_2017_2022.json', 'r') as f:
    contents = f.read()

# Split the contents into a list of individual JSON objects
records = contents.split('\n')

# Iterate over each record and extract the data you want
for record in records:
    # Skip empty records
    if not record:
        continue
    # Load the JSON object
    try:
        data = json.loads(record)
    except json.JSONDecodeError as e:
        print(f"Error while parsing JSON: {e}")
        continue
    # Extract the data you want
    headline = data.get('headline')
    short_description = data.get('short_description')
```

ii. Perform lexical analyses to extract the separate words, and to fold them all to lower case.

In [ ]:

```
import nltk
from nltk.tokenize import word_tokenize
import string

# Additional punctuation characters to remove
additional_punctuation = '''!()-[]{};:``'"\,<>./?@#$%^&*_~'''

# Combine the string.punctuation property with the additional punctuation characters
punctuation = string.punctuation + additional_punctuation

# Open the file and read the contents
with open('NewsCategoryDataset_2017_2022.json', 'r') as f:
    contents = f.read()

# Split the contents into a list of individual JSON objects
records = contents.split('\n')

# Iterate over each record and extract the data you want
for record in records:
    # Skip empty records
    if not record:
        continue
    # Load the JSON object
    try:
        data = json.loads(record)
    except json.JSONDecodeError as e:
        print(f"Error while parsing JSON: {e}")
        continue
    # Extract the data you want
    headline = data.get('headline')
    short_description = data.get('short_description')

    # Tokenize the headline and short description strings
    headline_tokens = word_tokenize(headline)
    short_description_tokens = word_tokenize(short_description)

     # Remove punctuation from the lists of tokens
    headline_tokens = [token for token in headline_tokens if token not in punctuation]
    short_description_tokens = [token for token in short_description_tokens if token not in punctuation]

    # Fold the tokens to lower case
    headline_tokens_lower = [token.lower() for token in headline_tokens]
    short_description_tokens_lower = [token.lower() for token in short_description_tokens]

    print("\nHeadline: \n" ,headline_tokens_lower)
    print("\nShort_description: \n" ,short_description_tokens_lower)
    print("\n----------------------------------------------------------------------------------------------------------------------------------------")
```

iii. Use a standard stop-word list for English to filter out the stop words.

- In this example, the nltk.corpus.stopwords.words() function is used to get a list of English stop words. The stop words are removed from the list of tokens using a list comprehension.

In [ ]:

```
from nltk.corpus import stopwords

# Additional punctuation characters to remove
additional_punctuation = '''!()-[]{};:``'"\,<>./?@#$%^&*_~'''

# Get a list of English stop words
stop_words = stopwords.words('english')

# Combine the string.punctuation property with the additional punctuation characters
punctuation = string.punctuation + additional_punctuation

# Open the file and read the contents
with open('NewsCategoryDataset_2017_2022.json', 'r') as f:
    contents = f.read()

# Split the contents into a list of individual JSON objects
records = contents.split('\n')

# Iterate over each record and extract the data you want
for record in records:
    # Skip empty records
    if not record:
        continue
    # Load the JSON object
    try:
        data = json.loads(record)
    except json.JSONDecodeError as e:
        print(f"Error while parsing JSON: {e}")
        continue
    # Extract the data you want
    headline = data.get('headline')
    short_description = data.get('short_description')

    # Tokenize the headline and short description strings
    headline_tokens = word_tokenize(headline)
    short_description_tokens = word_tokenize(short_description)

     # Remove punctuation from the lists of tokens
    headline_tokens = [token for token in headline_tokens if token not in punctuation]
    short_description_tokens = [token for token in short_description_tokens if token not in punctuation]

    # Fold the tokens to lower case
    headline_tokens_lower = [token.lower() for token in headline_tokens]
    short_description_tokens_lower = [token.lower() for token in short_description_tokens]

    # Remove stop words from the list of headline tokens
    headline_tokens_stop_words = [token for token in headline_tokens_lower if token not in stop_words]

    # Remove stop words from the list of short description tokens
    short_description_tokens_stop_words = [token for token in short_description_tokens_lower if token not in stop_words]

    print("\nHeadline: \n" ,headline_tokens_stop_words)
    print("\nShort_description: \n" ,short_description_tokens_stop_words)
    print("\n----------------------------------------------------------------------------------------------------------------------------------------")
```

iv. Use an implementation of Porter's stemmer to reduce terms to their stems (note that you may find a ready-made implementation provided that you reference its source).

> The Porter stemming algorithm (or 'Porter stemmer') is a process for removing the commoner morphological and inflexional endings from words in English. Its main use is as part of a term normalisation process that is usually done when setting up Information Retrieval systems. [Martin, P. (1999). The Porter Stemming Algorithm. [Online]. Available: https://tartarus.org/martin/PorterStemmer (https://tartarus.org/martin/PorterStemmer)]

In [ ]:

```
from nltk.stem.porter import PorterStemmer

# Additional punctuation characters to remove
additional_punctuation = '''!()-[]{};:``'"\,<>./?@#$%^&*_~'''

# Get a list of English stop words
stop_words = stopwords.words('english')

# Combine the string.punctuation property with the additional punctuation characters
punctuation = string.punctuation + additional_punctuation

# Open the file and read the contents
with open('NewsCategoryDataset_2017_2022.json', 'r') as f:
    contents = f.read()

# Split the contents into a list of individual JSON objects
records = contents.split('\n')

# Create an instance of the PorterStemmer class
stemmer = PorterStemmer()

# Iterate over each record and extract the data you want
for record in records:
    # Skip empty records
    if not record:
        continue
    # Load the JSON object
    try:
        data = json.loads(record)
    except json.JSONDecodeError as e:
        print(f"Error while parsing JSON: {e}")
        continue

    # Extract the data you want
    headline = data.get('headline')
    short_description = data.get('short_description')

    # Tokenize the headline and short description strings
    headline_tokens = word_tokenize(headline)
    short_description_tokens = word_tokenize(short_description)

     # Remove punctuation from the lists of tokens
    headline_tokens = [token for token in headline_tokens if token not in punctuation]
    short_description_tokens = [token for token in short_description_tokens if token not in punctuation]

    # Fold the tokens to lower case
    headline_tokens_lower = [token.lower() for token in headline_tokens]
    short_description_tokens_lower = [token.lower() for token in short_description_tokens]

    # Remove stop words from the list of headline tokens
    headline_tokens_filtered = [token for token in headline_tokens_lower if token not in stop_words]
    # Remove stop words from the list of short description tokens
    short_description_tokens_filtered = [token for token in short_description_tokens_lower if token not in stop_words]

    # Stem the filtered headline tokens
    headline_tokens_stemmed = [stemmer.stem(token) for token in headline_tokens_filtered]
    # Stem the filtered short description tokens
    short_description_tokens_stemmed = [stemmer.stem(token) for token in short_description_tokens_filtered]

    print("\nHeadline: \n" ,headline_tokens_stemmed)
    print("\nShort_description: \n" ,short_description_tokens_stemmed)
    print("\n----------------------------------------------------------------------------------------------------------------------------------------")
```

b. Calculate term weights using TF.IDF. Each headline record should be considered as a single document.

In [ ]:

```
import math
import numpy as np

def compute_tfidf(documents):
    # Tokenize the documents
    tokens = [word_tokenize(doc) for doc in documents]
    # Compute the term frequency (TF) for each term
    tf = []
    for doc in tokens:
        term_freq = {}
        for term in doc:
            if term in term_freq:
                term_freq[term] += 1
            else:
                term_freq[term] = 1
        tf.append(term_freq)
    # Compute the inverse document frequency (IDF) for each term
    idf = {}
    num_docs = len(documents)
    for doc in tokens:
        for term in doc:
            if term in idf:
                continue
            df = sum(1 for d in tokens if term in d)
            idf[term] = math.log(num_docs / df)
    # Compute the TF-IDF weight for each term
    tfidf = []
    for doc in tf:
        doc_tfidf = {}
        for term, freq in doc.items():
            doc_tfidf[term] = freq * idf[term]
        tfidf.append(doc_tfidf)
    # Create the term-by-document matrix
    terms = sorted(idf.keys())
    term_by_doc_matrix = np.zeros((len(terms), num_docs))
    for i, term in enumerate(terms):
        for j, doc in enumerate(tfidf):
            if term in doc:
                term_by_doc_matrix[i, j] = doc[term]
    return terms, term_by_doc_matrix

# Open the file and read the contents
with open('NewsCategoryDataset_2017_2022.json', 'r') as f:
    contents = f.read()

# Split the contents into a list of individual JSON objects
records = contents.split('\n')

# Create an instance of the PorterStemmer class
stemmer = PorterStemmer()

# Create a list to store the headlines
headlines = []

# Iterate over each record and extract the data you want
for record in records:
    # Skip empty records
    if not record:
        continue
    # Load the JSON object
    try:
        data = json.loads(record)
    except json.JSONDecodeError as e:
        print(f"Error while parsing JSON: {e}")
        continue
    # Extract the data you want
    headline = data.get('headline')
    short_description = data.get('short_description')

    # Tokenize the headline and short description strings
    headline_tokens = word_tokenize(headline)
    short_description_tokens = word_tokenize(short_description)

     # Remove punctuation from the lists of tokens
    headline_tokens = [token for token in headline_tokens if token not in punctuation]
    short_description_tokens = [token for token in short_description_tokens if token not in punctuation]

    # Fold the tokens to lower case
    headline_tokens_lower = [token.lower() for token in headline_tokens]
    short_description_tokens_lower = [token.lower() for token in short_description_tokens]

    # Remove stop words from the list of headline tokens
    headline_tokens_filtered = [token for token in headline_tokens_lower if token not in stop_words]

    # Remove stop words from the list of short description tokens
    short_description_tokens_filtered = [token for token in short_description_tokens_lower if token not in stop_words]

    # Stem the filtered headline tokens
    headline_tokens_stemmed = [stemmer.stem(token) for token in headline_tokens_filtered]
```

```
    # Stem the filtered short description tokens
    short_description_tokens_stemmed = [stemmer.stem(token) for token in short_description_tokens_filtered]

    # Join the stemmed tokens into a single string
    headline_stemmed = ' '.join(headline_tokens_stemmed)
    # Add the stemmed headline to the list of headlines
    headlines.append(headline_stemmed)

# Calculate the term weights using the compute_tfidf function
terms, term_by_doc_matrix = compute_tfidf(headlines)

for i, row in enumerate(term_by_doc_matrix):
    term = terms[i]
    print(f'TF-IDF weights for term "{term}": {row}')
    print('\n')
```

c. Extract the highest-weighted n% of the terms for each headline category (each JSON record has a field called “category”).

i. Calculate the average term weight for all terms over the documents within each category.

In [ ]:

```
from collections import Counter, defaultdict
from statistics import mean

# Open the file and read the contents
with open('NewsCategoryDataset_2017_2022.json', 'r') as f:
    contents = f.read()

# Split the contents into a list of individual JSON objects
records = contents.split('\n')

category_data = {}

# Create a list to store the headlines
categories = []

# Iterate over each record and extract the data you want
for record in records:
    # Skip empty records
    if not record:
        continue
    # Load the JSON object
    try:
        data = json.loads(record)
    except json.JSONDecodeError as e:
        print(f"Error while parsing JSON: {e}")
        continue

    # Extract the data you want
    category = data.get('category')

    # Tokenize the headline and category strings
    category_tokens = word_tokenize(category)

    # Remove punctuation from the lists of tokens
    category_tokens = [token for token in category_tokens if token not in punctuation]

    # Fold the tokens to lower case
    category_tokens_lower = [token.lower() for token in category_tokens]

    # Remove stop words from the list of category tokens
    category_tokens_filtered = [token for token in category_tokens_lower if token not in stop_words]

    # Stem the filtered category tokens
    category_tokens_stemmed = [stemmer.stem(token) for token in category_tokens_filtered]

    print("\nCategory: \n" ,category_tokens_stemmed)
    print("\n----------------------------------------------------------------------------------------------------------------------------------")

    # Join the stemmed tokens into a single string
    category_stemmed = ' '.join(category_tokens_stemmed)

    # Add the stemmed headline to the list of headlines
    categories.append(category_stemmed)

# Calculate the term weights using the compute_tfidf function
terms, term_by_doc_matrix = compute_tfidf(categories)

for i, row in enumerate(term_by_doc_matrix):
    term = terms[i]
    #print(f'TF-IDF weights for term "{term}": {row}')
    #print('\n')

# Initialize the dictionary to store the average term weights
average_term_weights = {}

# Initialize the defaultdict to store the sum and count of weights for each category
average_term_weights = defaultdict(Counter)

# Iterate over each row in the term-by-document matrix
for i, row in enumerate(term_by_doc_matrix):
    term = terms[i]
    # Update the sum and count of weights for each category
    for j, weight in enumerate(row):
        category = categories[j]
        average_term_weights[category]['sum'] += weight
        average_term_weights[category]['count'] += 1

# Compute the average term weights for each category
for category, weights in average_term_weights.items():
    average_weight = [weights['sum'] / weights['count']]
    average_weight = mean(average_weight)
    print(f'Average term weights for category "{category}": {average_weight}')
```

ii. Get the highest-weighted n% of the terms for each category. This list of terms, and their corresponding weights will subsequently be used to build a category keyword-cloud. This keyword-cloud will show what concepts each category generally mentions. n can be determined arbitrarily so that the keyword-cloud will contain neither too much nor too few words.

In [ ]:

```
# Tokenize the category strings
category_tokens = [word_tokenize(category) for category in categories]

# Remove punctuation from the lists of tokens
category_tokens = [[token for token in tokens if token not in punctuation] for tokens in category_tokens]

# Fold the tokens to lower case
category_tokens_lower = [[token.lower() for token in tokens] for tokens in category_tokens]

# Remove stop words from the list of category tokens
category_tokens_filtered = [[token for token in tokens if token not in stop_words] for tokens in category_tokens_lower]

# Stem the filtered category tokens
category_tokens_stemmed = [[stemmer.stem(token) for token in tokens] for tokens in category_tokens_filtered]

# Join the stemmed tokens into a single string
categories_stemmed = [' '.join(tokens) for tokens in category_tokens_stemmed]

# Calculate the term weights using the compute_tfidf function
terms, term_by_doc_matrix = compute_tfidf(categories_stemmed)

# Initialize a dictionary to store the top terms for each category
category_top_terms = {}

def get_top_n_percent_terms(terms, weights, n):
    # Sort the list of terms in descending order by their weights
    sorted_terms = sorted(range(len(terms)), key=lambda i: weights[i], reverse=True)

    # Calculate the number of terms to select as the top n percent
    num_terms = round(len(terms) * n / 100)

    # Return the top n percent of terms
    return [terms[i] for i in sorted_terms[:num_terms]]

# Initialize a dictionary to store the top terms for each category
category_top_terms = {}

# Iterate over the list of categories
for i, category in enumerate(categories):
    # Get the top 40% of terms for the category
    top_terms = get_top_n_percent_terms(terms, term_by_doc_matrix[:, i], 40)

    # Store the top terms in the dictionary
    category_top_terms[category] = top_terms

# Print the top terms for each category
for category, top_terms in category_top_terms.items():
    print(f"Top terms for category '{category}': {top_terms}")
```

iii. Extract the details of each category (including the category name, the list of articles in it, and list of highest-weighted terms for each category) as JSON. This will be used in the visualisation application described below.

In [ ]:

```
import json

# Open the file and read the contents
with open('NewsCategoryDataset_2017_2022.json', 'r') as f:
    contents = f.read()

# Split the contents into a list of individual JSON objects
records = contents.split('\n')

category_data = {}

# Iterate over each record and extract the data you want
for record in records:
    # Skip empty records
    if not record:
        continue
    # Load the JSON object
    try:
        data = json.loads(record)
    except json.JSONDecodeError as e:
        print(f"Error while parsing JSON: {e}")
        continue

    # Extract the data you want
    category = data.get('category')
    link = data.get('link')

    # Tokenize the category string
    category_tokens = word_tokenize(category)

    # Remove punctuation from the list of tokens
    category_tokens = [token for token in category_tokens if token not in punctuation]

    # Fold the tokens to lower case
    category_tokens_lower = [token.lower() for token in category_tokens]

    # Remove stop words from the list of tokens
    category_tokens_filtered = [token for token in category_tokens_lower if token not in stop_words]

    # Stem the filtered tokens
    category_tokens_stemmed = [stemmer.stem(token) for token in category_tokens_filtered]

    # Join the stemmed tokens into a single string
    category_stemmed = ' '.join(category_tokens_stemmed)

    # Get the top 40% of terms for the category
    top_terms = get_top_n_percent_terms(terms, term_by_doc_matrix[:, i], 40)

    # Create a dictionary to store the category details
    category_details = {
        'category': category,
        'links': [link],
        'top_terms': top_terms
    }

    # Check if the category is already in the category data dictionary
    if category_stemmed in category_data:
        # Update the links and top terms for the category
        category_data[category_stemmed]['links'].append(link)
        category_data[category_stemmed]['top_terms'] = top_terms
    else:
        # Add the category details to the category data dictionary
        category_data[category_stemmed] = category_details

# Open the file to write the JSON data
with open('category_data.json', 'w') as f:
    # Write the JSON data to the file with indentation
    json.dump(category_data, f, indent=2)

print('Output in category_data.json')
```

d. Use the document vectors to cluster the news headlines using the k-means algorithm. The choice of k is up to you. Note that you only need to do a single level of clustering, that is, no hierarchies are being requested.

In [ ]:

```
from sklearn.cluster import KMeans

def cluster_headlines(term_by_doc_matrix, num_clusters):
    # Transpose the term-by-document matrix to create a document-by-term matrix
    doc_by_term_matrix = term_by_doc_matrix.T
    # Convert the document-by-term matrix to a dense array
    doc_by_term_array = doc_by_term_matrix
    # Initialize the k-means model
    kmeans = KMeans(n_clusters=num_clusters)
    # Fit the model to the document vectors
    kmeans.fit(doc_by_term_array)
    # Predict the cluster labels for each document
    labels = kmeans.predict(doc_by_term_array)
    return labels

#Open the file and read the contents
with open('NewsCategoryDataset_2017_2022.json', 'r') as f:
    contents = f.read()

# Split the contents into a list of individual JSON objects
records = contents.split('\n')

headline_data = {}

# Create a list to store the headlines
headlines = []

# Iterate over each record and extract the data you want
for record in records:
    # Skip empty records
    if not record:
        continue
    # Load the JSON object
    try:
        data = json.loads(record)
    except json.JSONDecodeError as e:
        print(f"Error while parsing JSON: {e}")
        continue

    # Extract the data you want
    headline = data.get('headline')

    # Tokenize the headline and category strings
    headline_tokens = word_tokenize(headline)

    # Remove punctuation from the lists of tokens
    headline_tokens = [token for token in headline_tokens if token not in punctuation]

    # Fold the tokens to lower case
    headline_tokens_lower = [token.lower() for token in headline_tokens]

    # Remove stop words from the list of category tokens
    headline_tokens_filtered = [token for token in headline_tokens_lower if token not in stop_words]

    # Stem the filtered category tokens
    headline_tokens_stemmed = [stemmer.stem(token) for token in headline_tokens_filtered]

    print("headline: \n" ,headline_tokens_stemmed)
    print("\n----------------------------------------------------------------------------------------------------------------------------------------------------------")

    # Join the stemmed tokens into a single string
    headline_stemmed = ' '.join(headline_tokens_stemmed)

    # Add the stemmed headline to the list of headlines
    headlines.append(headline_stemmed)

# Calculate the term weights using the compute_tfidf function
terms, term_by_doc_matrix = compute_tfidf(headlines)

for i, row in enumerate(term_by_doc_matrix):
    term = terms[i]
    print(f'TF-IDF weights for term "{term}": {row}')
    print('\n')

# Cluster the headlines
num_clusters = 5
labels = cluster_headlines(term_by_doc_matrix, num_clusters=num_clusters)

# Group the headlines by cluster
clustered_headlines = {}
for i, label in enumerate(labels):
    if label not in clustered_headlines:
        clustered_headlines[label] = []
    clustered_headlines[label].append(headlines[i])

#Print the headlines in each cluster
for label, cluster in clustered_headlines.items():
    print(f'Cluster {label}:')
    for headline in cluster:
        print(headline)
```

```
    print()
```

In [ ]:

e. Extract the highest-weighted n% of the terms for each cluster generated in the previous step.

i. Calculate the average term weight for all terms over the documents within each cluster

In [ ]:

```
# Transpose the term-by-document matrix to create a document-by-term matrix
doc_by_term_matrix = term_by_doc_matrix.T

# Create a list to store the average term weights for each cluster
avg_term_weights = []

# Iterate over each cluster label
for label in np.unique(labels):
  # Select the documents in the document-by-term matrix that belong to the cluster
  cluster_docs = doc_by_term_matrix[labels == label]
  # Calculate the sum of the term weights for each term over the selected documents
  term_weights_sum = cluster_docs.sum(axis=0)
  # Calculate the average term weight for each term in the cluster
  avg_term_weight = term_weights_sum / cluster_docs.shape[0]
  # Add the average term weights for each term in the cluster to the list
  avg_term_weights.append(avg_term_weight)

# Print the average term weights for each cluster
for cluster_idx, avg_term_weight in enumerate(avg_term_weights):
    print(f'Cluster {cluster_idx}:')
    for term_idx, avg_weight in enumerate(avg_term_weight):
        term = terms[term_idx]
        print(f'  Term "{term}": {avg_weight: }')
    print()
```

In [ ]:

ii. Get the highest-weighted n% of the terms for each cluster. This list of terms, and their corresponding weights will subsequently be used to build a cluster keyword-cloud. This keyword-cloud will show what concepts each cluster generally mentions. n can be determined arbitrarily so that the keyword-cloud will contain neither too much nor too few words.

In [ ]:

```
from collections import defaultdict

def get_top_n_percent_terms(labels, terms, weights, n):
    # Initialize a dictionary to store the top terms and weights for each cluster
    top_terms = defaultdict(list)

    # Iterate over the list of labels
    for label in set(labels):
        # Extract the term weights for the cluster
        cluster_weights = weights[:, labels == label]
        # Calculate the average term weights for the cluster
        avg_weights = cluster_weights.mean(axis=1)
        # Sort the list of terms in descending order by their average weights
        sorted_terms = sorted(range(len(terms)), key=lambda i: avg_weights[i], reverse=True)

        # Calculate the number of terms to select as the top n percent
        num_terms = round(len(terms) * n / 100)

        # Extract the top n percent of terms for the cluster
        cluster_top_terms = [terms[i] for i in sorted_terms[:num_terms]]

        # Extract the corresponding weights for the top terms
        cluster_top_weights = [avg_weights[i] for i in sorted_terms[:num_terms]]

        # Store the top terms and weights in the dictionary
        top_terms[label] = list(zip(cluster_top_terms, cluster_top_weights))

    return top_terms

# Get the top 40% of terms for each cluster
n = 40
cluster_top_terms = get_top_n_percent_terms(labels, terms, term_by_doc_matrix, n)

# Print the top terms and weights for each cluster
for label, top_terms in cluster_top_terms.items():
    print(f'Cluster {label}:')
    for term, weight in top_terms:
        print(f'  {term}: {weight: }')
    print()
```

iii. Extract the details of each cluster (the cluster ID, the list of articles in it, and list of highest-weighted terms for each cluster) as JSON. This will be used in the visualisation application described below.

In [ ]:

```
import json

# Get the cluster labels and the documents in each cluster
cluster_labels = labels
cluster_docs = term_by_doc_matrix

# Load the JSON file containing the links for each document
try:
    with open('NewsCategoryDataset_2017_2022.json', 'r') as f:
        doc_links = json.load(f)
except FileNotFoundError:
    #print("The file 'NewsCategoryDataset_2017_2022.json' was not found.")
    doc_links = []
except json.JSONDecodeError:
    #print("There was an error decoding the JSON file.")
    doc_links = []

# Get the top 40% of terms for each cluster
n = 40
cluster_top_terms = get_top_n_percent_terms(cluster_labels, terms, cluster_docs, n)

# Initialize an empty dictionary to store the cluster details
cluster_data = {}

# Iterate over the cluster labels
for label in set(cluster_labels):
  # Initialize an empty list to store the links for the cluster
  cluster_links = []
  # Iterate over the links for each document
  for doc_idx, doc_link in enumerate(doc_links):
    # Check if the document belongs to the cluster
    if cluster_labels[doc_idx] == label:
      # If it does, add the link to the list
      cluster_links.append(doc_link['link'])
  # Extract the top terms and weights for the cluster
  cluster_top_term_weights = cluster_top_terms[label]
  # Create a dictionary to store the cluster details
  cluster_details = {
    'cluster_id': str(label),
    'links': cluster_links,
    'top_terms': cluster_top_term_weights
  }
  # Add the cluster details to the cluster data dictionary
  cluster_data[str(label)] = cluster_details

# Open the file to write the JSON data
try:
    with open('cluster_data.json', 'w') as f:
      # Write the JSON data to the file with indentation
        json.dump(cluster_data, f, indent=2)
    print('Output written to cluster_data.json')
except IOError:

    print("An error occurred while writing to the file.")
```

Instructions on how to start the Flask Application:

1. Open the "*web_application*" folder in a terminal or command prompt and navigate to the directory where the file "*app.py*" is saved.

2. Run the following command (or run from the IDE), to start the Flask development server: python app.py.

3. After the server starts, you can view the web pages by navigating to http://localhost:5000/ (http://localhost:5000/) in your web browser.

4. After navigating to the port, a web page with two clickable links will be shown.

5. First link will show the Category data. The dropdown menu can be used to choose to view information about a specific category.There is aswell the bubble chart which when a bubble is clicked will show inofrmation about a specific category.

6. On the other hand, the second link will show the Cluster data. The dropdown menu can be used to view information about a specific cluster.

Other Notes:

- **The category_data.json and cluster_data.json files must be in the same directory as the script for the code to work correctly.**
- **The HTML files are found in the templates folder.**
- **The D3.js library was used to implement the Bubble Charts.**

Difficulties:

1. The bubble chart for the list of categories (question 2,c) was implemented and the categories' information is outputted when the bubbles are clicked on, but for some reason the bubbles are too zoomed in and only two bubbles can be seen. Eventhough the other bubbles are there, I could not get the correct parameters to show them all.

2. The code for the bubble chart for the clusters (question 2,d) could not work.

Python Flask Server

In [ ]:

```
import json
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/')
def index():
    return render_template('index.html')

@app.route('/category_data')
def category_data():
    with open('category_data.json', 'r') as f:
        data = json.load(f)
    return render_template('category_data.html', data=data)

@app.route('/cluster_data')
def cluster_data():
    with open('cluster_data.json', 'r') as f:
        data = json.load(f)
    return render_template('cluster_data.html', data=data)

if __name__ == '__main__':
    app.run()
```

index.html

In [ ]:

```
<!doctype html>
<html>
    <head>
        <title>JSON Data</title>
    </head>
    <body>
        <h1>JSON Data</h1>
        <ul>
            <li><a href="/category_data">Category Data</a></li>
            <li><a href="/cluster_data">Cluster Data</a></li>
        </ul>
    </body>
</html>
```

category_data.json

In [ ]:

```
<!doctype html>
<html>
  <head>
    <title>Category Data</title>
    <script src="https://d3js.org/d3.v5.min.js"></script>
  </head>
  <body>
    <h1>Category Data</h1>
    <form>
      <label for="category-select">Select a category:</label><br>
      <select id="category-select" onchange="displayCategory(this.value)">
        {% for category, data in data.items() %}
          <option value="{{ category }}">{{ category }}</option>
        {% endfor %}
      </select>
    </form>
    <br>
    <div id="category-info">
    <script>
      function displayCategory(category) {
        var data = {{ data | tojson }};
        var info = data[category];
        var html = '<h2>' + info.category + '</h2>';
        html += '<h3>Links:</h3>';
        html += '<ul>';
        for (var i = 0; i < info.links.length; i++) {
          html += '<li><a href="' + info.links[i] + '">' + info.links[i] + '</a></li>';
        }
        html += '</ul>';
        html += '<h3>Top Terms:</h3>';
        html += '<ul>';
        for (var i = 0; i < info.top_terms.length; i++) {
          html += '<li>' + info.top_terms[i] + '</li>';
        }
        html += '</ul>';
        document.getElementById('category-info').innerHTML
        = html;
    }
    </script>
    </div>

    <!-- Bubble Chart -->
    <div id="bubble-chart">
    <h2> Category Bubble Chart: </h2>
    <script>
      function displayCategory(category) {
        var data = {{ data | tojson }};
        var info = data[category];
        var html = '<h2>' + info.category + '</h2>';
        html += '<h3>Links:</h3>';
        html += '<ul>';
        for (var i = 0; i < info.links.length; i++) {
          html += '<li><a href="' + info.links[i] + '">' + info.links[i] + '</a></li>';
        }
        html += '</ul>';
        html += '<h3>Top Terms:</h3>';
        html += '<ul>';
        for (var i = 0; i < info.top_terms.length; i++) {
          html += '<li>' + info.top_terms[i] + '</li>';
        }
        html += '</ul>';
        document.getElementById('category-info').innerHTML
        = html;
      }

      // Create Bubble Chart
      var bubbleData = {{ data | tojson }};

      var width = 600;
      var height = 600;

      var svg = d3.select('#bubble-chart')
        .append('svg')
        .attr('width', width)
        .attr('height', height);

      var color = d3.scaleOrdinal(d3.schemeCategory10);

      var simulation = d3.forceSimulation(Object.keys(bubbleData).map(function(d) {
        return {
          name: d,
          value: bubbleData[d].links.length,
          links: bubbleData[d].links,
          top_terms: bubbleData[d].top_terms
        };
      }))
        .force('charge', d3.forceManyBody().strength(-5))
        .force('center', d3.forceCenter(width / 2, height / 2))
        .force('x', d3.forceX().x(function(d) {
          return d.value * 10;
```

```
      }))
      .force('y', d3.forceY().y(height / 2))
      .on('tick', ticked);

    var circles = svg.selectAll('circle')
      .data(simulation.nodes())
      .enter()
      .append('circle')
      .attr('r', function(d) {
        return d.value;
      })
      .attr('fill', function(d) {
        return color(d.name);
      })
      .on('click', function(d) {
        var html = '<h2>' + d.name + '</h2>';
        html += '<h3>Links:</h3>';
        html += '<ul>';
        for (var i = 0; i < d.links.length; i++) {
          html += '<li><a href="' + d.links[i] + '">' + d.links[i] + '</a></li>';
        }
        html += '</ul>';
        html += '<h3>Top Terms:</h3>';
        html += '<ul>';
        for (var i = 0; i < d.top_terms.length; i++) {
          html += '<li>' + d.top_terms[i] + '</li>';
        }
        html += '</ul>';
        document.getElementById('category-info').innerHTML = html;
      });

    function ticked() {
      circles
        .attr('cx', function(d) {
          return d.x;
        })
        .attr('cy', function(d) {
          return d.y;
        });
    }

  </script>
 </div>
 </body>
</html>
```

cluster_data.json

In [ ]:

```
<!doctype html>
<html>
  <head>
    <title>Cluster Data</title>
  </head>
  <body>
    <h1>Cluster Data</h1>
    <form>
      <label for="cluster-select">Select a cluster:</label><br>
      <select id="cluster-select" onchange="displayCluster(this.value)">
        {% for cluster_id, data in data.items() %}
          <option value="{{ cluster_id }}">{{ cluster_id }}</option>
        {% endfor %}
      </select>
    </form>
    <br>
    <div id="cluster-info"></div>

    <script>
      function displayCluster(cluster_id) {
        var data = {{ data | tojson }};
        var info = data[cluster_id];
        var html = '<h2>Cluster ' + info.cluster_id + '</h2>';
        html += '<h3>Links:</h3>';
        html += '<ul>';
        for (var i = 0; i < info.links.length; i++) {
          html += '<li><a href="' + info.links[i] + '">' + info.links[i] + '</a></li>';
        }
        html += '</ul>';
        html += '<h3>Top Terms:</h3>';
        html += '<ul>';
        for (var i = 0; i < info.top_terms.length; i++) {
          html +='<li>' + info.top_terms[i][0] + ' (' + info.top_terms[i][1] + ')</li>';
        }
        html += '</ul>';
        document.getElementById('cluster-info').innerHTML
        = html;
        }
        </script>


    <!-- Bubble Chart --><!-- Bubble Chart -->
    <div id="bubble-chart"></div>

    <script src="https://d3js.org/d3.v5.min.js"></script>
    <script>
      var width = 800;
      var height = 600;
      var padding = 20;

      var svg = d3.select("#bubble-chart")
        .append("svg")
        .attr("width", width)
        .attr("height", height);

      function displayBubbleChart(data) {
        // Set up the scales
        var xScale = d3.scaleLinear()
          .domain([0, d3.max(data, function(d) { return d.x; })])
          .range([padding, width - padding]);

        var yScale = d3.scaleLinear()
          .domain([0, d3.max(data, function(d) { return d.y; })])
          .range([height - padding, padding]);

        var rScale = d3.scaleLinear()
          .domain([0, d3.max(data, function(d) { return d.r; })])
          .range([5, 30]);

        // Add the circles
        var circles = svg.selectAll("circle")
          .data(data)
          .enter()
          .append("circle")
          .attr("cx", function(d) { return xScale(d.x); })
          .attr("cy", function(d) { return yScale(d.y); })
          .attr("r", function(d) { return rScale(d.r); });
      }

      var data = {{ data | tojson }};
      var bubbleData = [];
      for (var cluster_id in data) {
        var cluster = data[cluster_id];
        bubbleData.push({
          x: cluster.x,
          y: cluster.y,
          r: cluster.r
        });
      }
```

```
        displayBubbleChart(bubbleData);
      </script>
    </body>
</html>
```