News Topics Prediction Using Web Scraping and Machine Learning Techniques

Haikun Guo
College of Science and Engineering
James Cook University
Cairns, Australia
haikun.guo@my.jcu.edu.au

1. Overview

The primary issue addressed in this report is news topic prediction by machine learning techniques. News topic prediction involves categorizing news articles into predefined topics or categories such as politics, sports, technology, health, and entertainment. This task is critical due to the vast amount of news content generated daily, which necessitates effective organization for easy access and retrieval.

News topic prediction is prominently presented on the internet across various platforms, including news websites, blogs, and social media. Major news outlets like BBC, CNN, and Reuters continuously publish a plethora of articles that need to be categorized accurately to maintain a structured and user-friendly interface.

Expanding the chosen domain could involve integrating advanced machine learning techniques (Sebastiani, F., 2002; McCallum, A., & Nigam, K., 1998) to refine the accuracy and efficiency of news classification systems (Conneau et al., 2017). This expansion might include developing multilingual classification systems to cater to a global audience, incorporating real-time classification capabilities to handle breaking news, and enhancing user personalization through more nuanced topic identification.

Machine learning models can learn from large datasets of labeled news articles to identify patterns and features that distinguish different topics. Several previous studies such as Convolutional Neural Networks (CNNs) (Conneau et al., 2017), Recurrent Neural Networks (RNNs), and Transformers (Devlin et al., 2019) have explored various approaches to this task.

2. Domains

For the purpose of this study, the following website URLs have been chosen to be crawled: The base url: https://www.voanews.com

According to the keywords in 'medical', 'technology', 'entertainment', 'sports', 'environment', 'politics', I specifically crawled the data from:
https://www.voanews.com/s?k=medical&tab=all&pi=1&r=any&pp=20
https://www.voanews.com/s?k=technology&tab=all&pi=1&r=any&pp=20

https://www.voanews.com/s?k=entertainment&tab=all&pi=1&r=any&pp=20
https://www.voanews.com/s?k=sports&tab=all&pi=1&r=any&pp=20
https://www.voanews.com/s?k=environment&tab=all&pi=1&r=any&pp=20
https://www.voanews.com/s?k=politics&tab=all&pi=1&r=any&pp=20

With the page number changing, the argument in the url "pi=1" changes from 1 to 5. I crawled 240 news articles from the base url and the corresponding branch urls in total.

These websites were selected for their extensive and varied coverage of global news, which ensures a rich dataset for news topic classification. They provide content across multiple domains relevant to the issue, such as politics, health, technology, and entertainment, offering a balanced view of different news categories.

For this study, it has been determined that it is permissible to crawl articles from VOA News (https://www.voanews.com). VOA News, being a publicly funded international broadcaster, provides its content freely accessible to the public. The website's terms of service (Terms of Use and Privacy Note - VOA - Voice of America English News (voanews.com)) do not explicitly prohibit web scraping for non-commercial, academic purposes, provided that the content is not used in a manner that would violate copyright laws or the website's terms and conditions.

While crawling VOA News, the data will be used strictly for academic purposes, ensuring that the content is not republished or redistributed in a way that violates copyright laws. Proper attribution will be given to VOA News for all content used in the study.

3. WebCrawler Workflow

3.1 Technologies

The requests library is used to send HTTP requests to web servers and retrieve HTML content. Alternatives include urllib and http.client. However, requests is more user-friendly and provides a more intuitive API for handling HTTP requests and responses.

BeautifulSoup is employed to parse the HTML content and extract data using its powerful parsing capabilities. While lxml and Scrapy also offer HTML parsing, BeautifulSoup is known for its ease of use and flexibility.

The pandas library is utilized to store and manipulate the extracted data in a DataFrame format. Alternatives like numpy can handle data storage, but pandas provides more functionality for handling tabular data and exporting to formats like CSV.

3.2 Domain Complexity

The domains I am using are news domains.The VOA News websites have public news database. After finding the class of the containers of all the searching results, I found all the news websites through hyper-reference link (href) extraction in the containers. The desired news urls resides within HTML tags <a>, and the desired contents of articles resides within tag <p>. The code segments are shown as Code 1.

```python
        # Send requests
        response = requests.get(self.link_to_explore)
        html_content = response.content

        # Parse the html contents
        soup = BeautifulSoup(html_content, "html.parser")

        container = soup.find(class_="media-block-wrap")

        # Get all the hrefs (repetition included)
        if container:
            links = container.find_all("a")
            links_exclude = container.find_all("a", class_ =
"links__item-link")
            links = list(set(links) - set(links_exclude))
        else:
            print("Container not found!")
            return
        for link in links:
            href = link.get("href")
            self.hrefs.add(href)
```

Code 1: URL Crawling

3.3 Crawler Optimization

Additionally, in order to ensure the portability of the crawler and to make sure the data extraction can be applied to any other websites, I designed the crawler as a my_crawler class.

The class is initialized with a starting URL and a keyword, along with setting up data structures for storing articles and tracking URLs.

In the class, the crawler constructs initial URLs for each page based on pagination parameters and sends HTTP requests to retrieve the HTML content and uses BeautifulSoup to parse it. Then, the crawler identifies and collects article URLs, excluding non-relevant links and repeated links using a set.

After fetching the URLs, the crawler iterates through the collected URLs, constructs the full URLs for the articles, and sends requests to fetch their content.

The article-extracting function (extract_data_at_once) includes rate limiting (sleep intervals, Code 2) to avoid overloading the server and to comply with potential access restrictions.

```
                # Control the frequency of the requests
                time_between_requests = 6 / 3  # 2 RPM
                time.sleep(time_between_requests)
```

Code 2: Sleep Intervals

The designed class doesn't involve time complexity or space complexity optimization, remaining a O(n) in both URL-crawling and article-crawling functions.

3.4 Data Storage

A DataFrame is used to store articles in a tabular format, making it easy to manipulate and analyze the data. Compared to simple lists or dictionaries, DataFrames offer more functionality for data handling.

The final data is exported to a CSV file for its simplicity and ease of use in the downstream task. I separated the articles from different topics for further fusion. Additionally, I utilized utf_8_sig as the encoding format to avoid the mojibakes in the CSV files. The specific code is shown as follows:

```
def save_data_to_file(self):
        self.articles.to_csv ('./corpus/%s_dataset.csv' % self.keyword,
index = None, header=True, encoding='utf_8_sig')
```

Code 3: Save data

4. Data Wrangling

4.1 Data Cleaning

The corpus I was using in the data wrangling section is the dataset of news articles crawled by the crawler class I mentioned in section 3.3. In order to clean the data and normalize the data dimensions, I employed methods include:

(1) Eliminate the HTML labels and entities
(2) Subtract punctuations, including the Unicode punctuations
(3) Lowercasing all the characters
(4) Subtract all the stop words
(5) Correct possible spelling errors
(6) Delete extra white spaces

The detailed code of data cleaning is shown in Code 4.

```
def clean_text(text):
    # Eliminate HTML lables and entities
    text = re.sub(r"<.*?>", "", text)
    # Subtract punctuations, including unicode characters
    # Create a translation table
    translator = str.maketrans(
        {
            "\u2018": "",
            "\u2019": "",  # Single quotes
            "\u201c": "",
            "\u201d": "",  # Double quotes
            "\u2026": "",
            "\u2013": "",
            "\u2014": "",  # Ellipsis and dashes
        }
    )
    translator.update(str.maketrans("", "", string.punctuation))
    text = text.translate(translator)
    # Lowercasing
    text = text.lower()

    # Subtract stop words
    stop_words = stopwords.words("english")
    stop_words.extend(
        [
            "from",
            "said",
            "say",
            "reported",
            "news",
            "according",
            "would",
            "could",
            "posting",
            "article",
            "one",
            "two",
            "three",
        ]
    )
    text = " ".join([word for word in text.split() if word not in
stop_words])

    # Correct possible spelling errors
    text = str(TextBlob(text).correct())

    # Detele extra white spaces
    text = re.sub(r"\s+", " ", text).strip()

    return text
```
Code 4: Data Cleaning

4.2 Feature Extracting

Word2Vec (Joachims, T., 1998) was employed to extract features from the articles. Specifically, CBOW word embedding was leveraged, all the articles were tokenized by word_tokenize from the package nltk, each token will then be embedded into a 100 dimensional vector. Additionally, the tokens that occur less than one time will be abandoned. The model configuration of the feature extraction is shown below:

```python
def word2vec_embedding(tokens):
    # Parameters:
    # - sg: 1 for skip-gram; 0 for CBOW, default = 0
    # - vector_size: dimensions of the word vectors
    # - window: context window size
    # - min_count: minimum frequency of words to consider
    model = Word2Vec(tokens, vector_size=100, window=5, min_count=1,
workers=4)
    return model
```

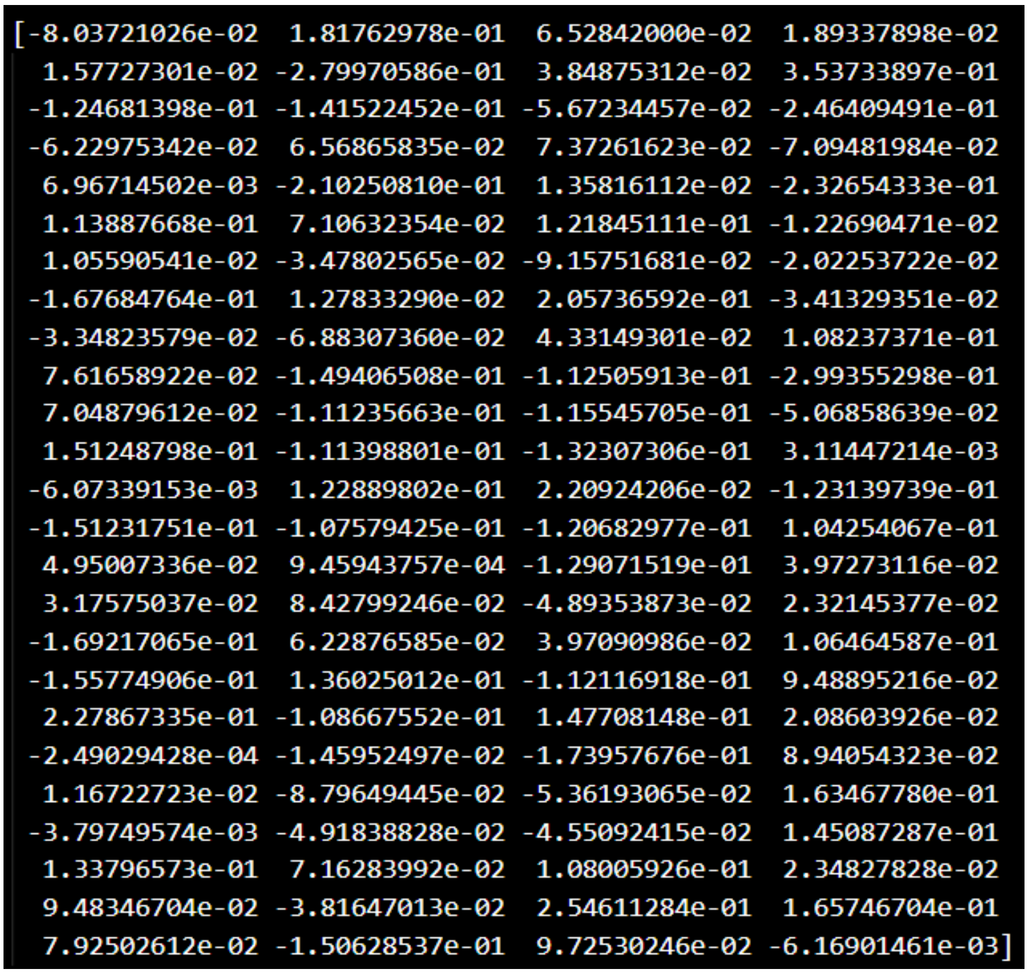
Code 5: Word2Vec Model

where tokens is a list of tokens from a single article, vector_size is set to 100 to generate a 100-dimension vector for each token, windows is set to 5 to set up the greatest distance between the current word and the predicted word, min_count is set to 1 to abandon the tokens that appears less than one time.

Other than the hyper-parameters given above, other hyper-parameters were set in default value in the Word2Vec API in gensim.models module, listed are some more important hyper-parameters:

(1) Original learning rate alpha: 0.025
(2) Thresholds for random sampling of high-frequency words sample: 1e-3
(3) Minimal learning rate min_alpha: 0.0001
(4) Algorithm sg: 0, set as CBOW
(5) Hierarchical softmax hs: 0, use negative sampling
(6) Number of noise words in negative sampling negative: 5
(7) Apply CBOW mean value or summation cbow_mean: 1, use mean value

For each article, the dimensions of their word embeddings will then be averaged, and for those cases where no word in the article is in the vocabulary, I set a 100-dimensional zero vector as the embeddings. In this way, each article was embedded into a 100-dimensional vector. This is a sample of a article vector:

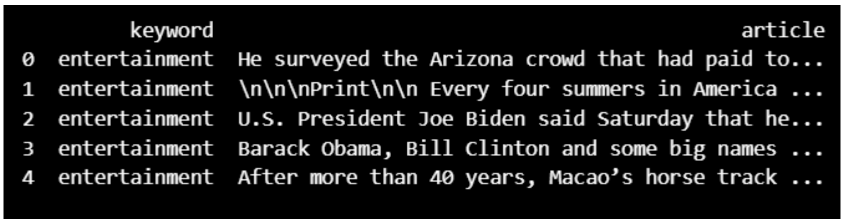


```
[-8.03721026e-02  1.81762978e-01  6.52842000e-02  1.89337898e-02
  1.57727301e-02 -2.79970586e-01  3.84875312e-02  3.53733897e-01
 -1.24681398e-01 -1.41522452e-01 -5.67234457e-02 -2.46409491e-01
 -6.22975342e-02  6.56865835e-02  7.37261623e-02 -7.09481984e-02
  6.96714502e-03 -2.10250810e-01  1.35816112e-02 -2.32654333e-01
  1.13887668e-01  7.10632354e-02  1.21845111e-01 -1.22690471e-02
  1.05590541e-02 -3.47802565e-02 -9.15751681e-02 -2.02253722e-02
 -1.67684764e-01  1.27833290e-02  2.05736592e-01 -3.41329351e-02
 -3.34823579e-02 -6.88307360e-02  4.33149301e-02  1.08237371e-01
  7.61658922e-02 -1.49406508e-01 -1.12505913e-01 -2.99355298e-01
  7.04879612e-02 -1.11235663e-01 -1.15545705e-01 -5.06858639e-02
  1.51248798e-01 -1.11398801e-01 -1.32307306e-01  3.11447214e-03
 -6.07339153e-03  1.22889802e-01  2.20924206e-02 -1.23139739e-01
 -1.51231751e-01 -1.07579425e-01 -1.20682977e-01  1.04254067e-01
  4.95007336e-02  9.45943757e-04 -1.29071519e-01  3.97273116e-02
  3.17575037e-02  8.42799246e-02 -4.89353873e-02  2.32145377e-02
 -1.69217065e-01  6.22876585e-02  3.97090986e-02  1.06464587e-01
 -1.55774906e-01  1.36025012e-01 -1.12116918e-01  9.48895216e-02
  2.27867335e-01 -1.08667552e-01  1.47708148e-01  2.08603926e-02
 -2.49029428e-04 -1.45952497e-02 -1.73957676e-01  8.94054323e-02
  1.16722723e-02 -8.79649445e-02 -5.36193065e-02  1.63467780e-01
 -3.79749574e-03 -4.91838828e-02 -4.55092415e-02  1.45087287e-01
  1.33796573e-01  7.16283992e-02  1.08005926e-01  2.34827828e-02
  9.48346704e-02 -3.81647013e-02  2.54611284e-01  1.65746704e-01
  7.92502612e-02 -1.50628537e-01  9.72530246e-02 -6.16901461e-03]
```

Figure 1: Article Embedding Sample

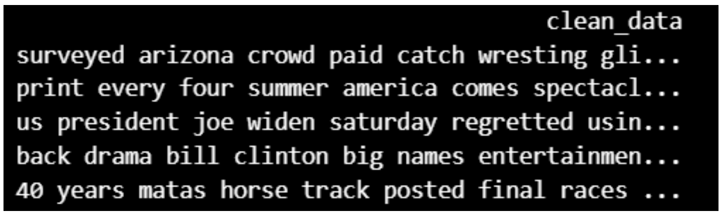
4.3 Data Summarization

As previously mentioned in Section 3.4, the dataset was saved in CSV format. The first five data in the corpus is shown below:



```
         keyword                                            article
0  entertainment  He surveyed the Arizona crowd that had paid to...
1  entertainment  \n\n\nPrint\n\n Every four summers in America ...
2  entertainment  U.S. President Joe Biden said Saturday that he...
3  entertainment  Barack Obama, Bill Clinton and some big names ...
4  entertainment  After more than 40 years, Macao’s horse track ...
```

Figure 2: Data Before Cleaning



```
                                       clean_data
surveyed arizona crowd paid catch wresting gli...
print every four summer america comes spectacl...
us president joe widen saturday regretted usin...
back drama bill clinton big names entertainmen...
40 years matas horse track posted final races ...
```

Figure 3: Data After Cleaning

The word frequency and 2-grams graph of the sample (corpus before lemmatizing) are shown as follows:

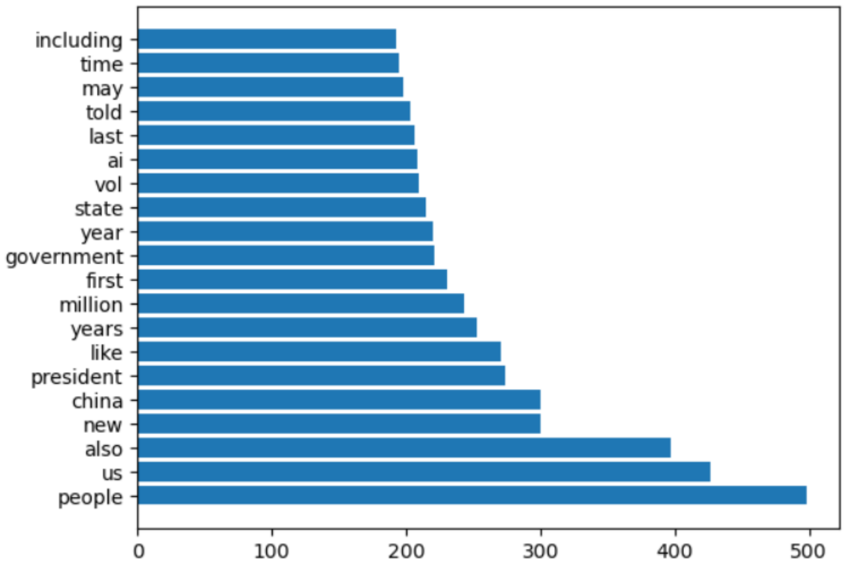


Figure 4: Word Frequency Bar Diagram

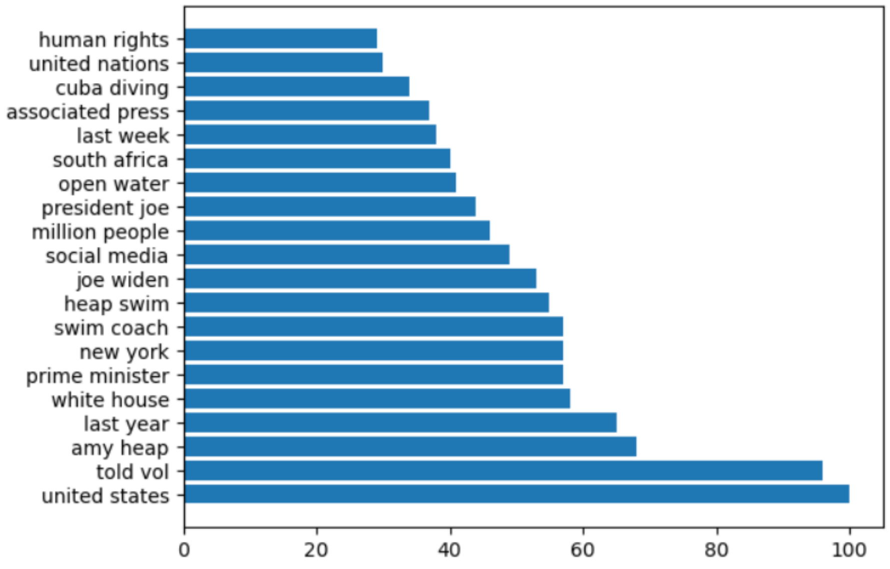


Figure 5: 2-grams Frequency Bar Diagram

From the word frequency bar diagram I can see that a considerable part of the high-frequency words are politics-related, this might due to the dataset bias, words like “people”, “china”, “government” show that the news are more likely to related to politics though the corpus also contains other topics such as “entertainment” and “environment”. Words like “united states”, “president joe”, and “new york” may due to the attribute of the base URL-VOA news is an American website.

Also, the wordcloud is shown as follows:

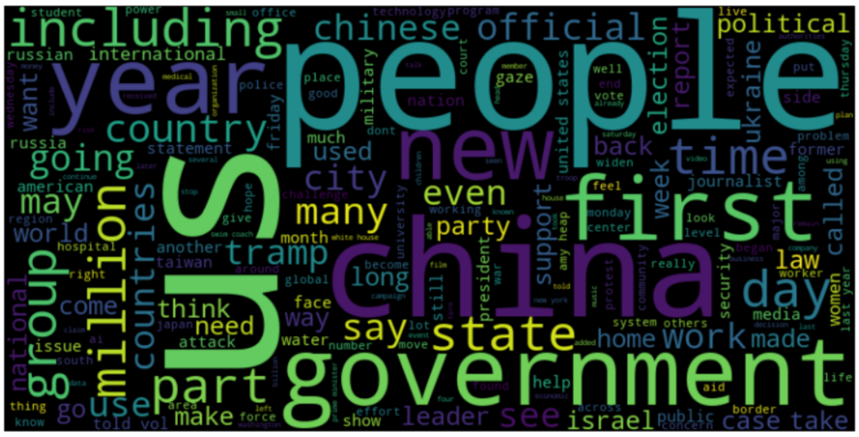


Figure 6: Wordcloud

The word frequency and 2-grams graph of the corpus (after lemmatizing) are shown as follows:

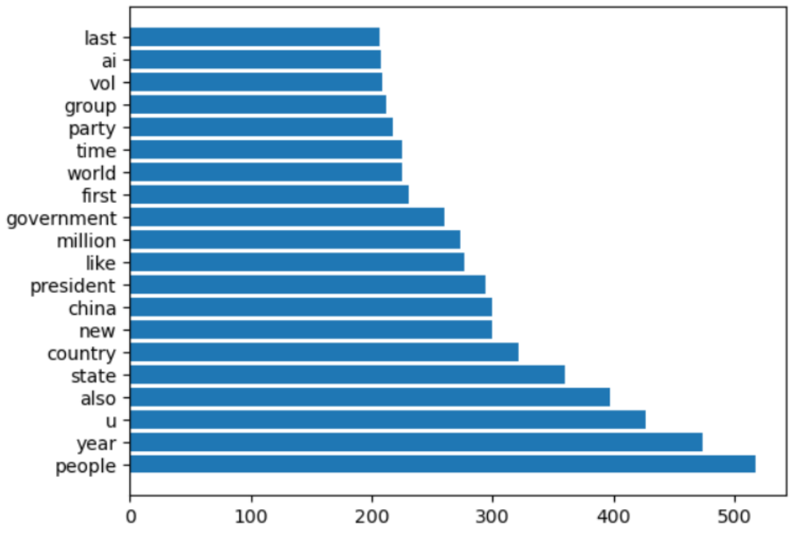


Figure 7: Word Frequency Bar Diagram

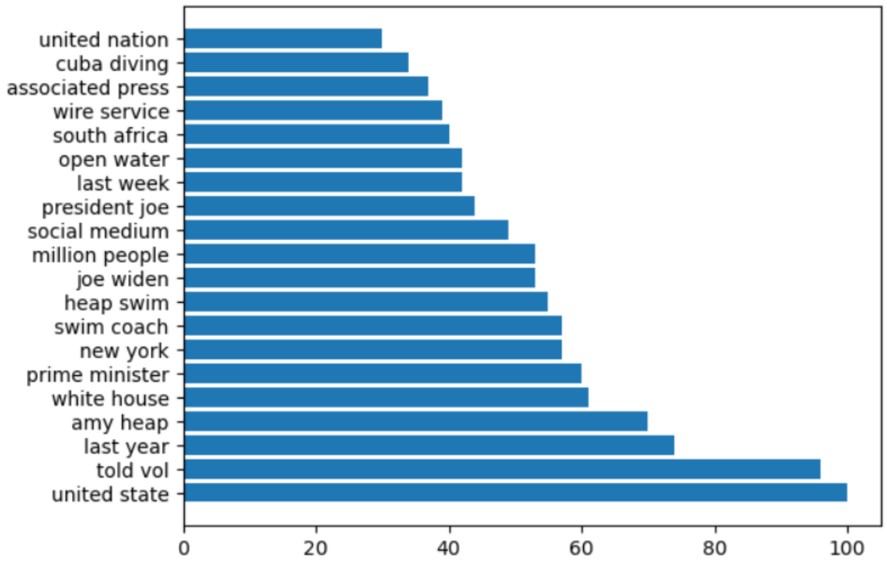


Figure 8: 2-grams Bar Diagram

After lemmatizing, the word frequencies were changed unexpectedly, words like “year” increases it’s frequency due to the lemmatization. Also, there are some words that has been mistakenly lemmatized, such as “us”. The wordcloud of the lemmatized corpus is shown as follows:

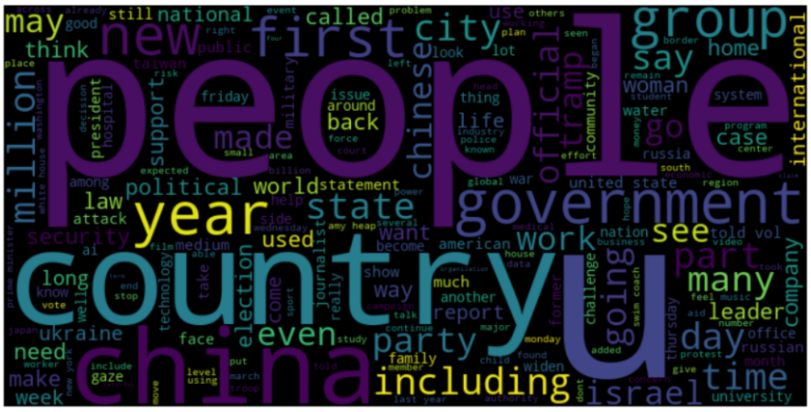


Figure 9: Wordcloud (lemmatized)

The dataset has its limitations such as website attribution, topic categories, and data size. For example, the topics I chose are 'medical', 'technology', 'entertainment', 'sports', 'environment', and 'politics', but most of them can be related to politics in some situations, this may lead to the bias that the most frequently appeared word are related to countries and other politic nouns.

5. Machine Learning

5.1 Structure

The dataset document is split into training and testing sets using train_test_split, ensuring that 30% of the data is reserved for testing, and the split is reproducible with a fixed random_state = 3.

A pipeline is created, which standardizes the data using StandardScaler and applies the LinearDiscriminantAnalysis model. Grid search is deployed for tuning the hyper-parameters. Specifically, I used GridSearchCV function to utilize k-fold cross-validation (Kohavi, R., 1995) on model training.

The k-fold cross validation is configured as follows:

```
# n_splits: Number of folds. Must be at least 2.
# shuffle: if shuffle the dataset
# random_state: When shuffle is True, random_state controls the
randomness of each fold.
kfold = KFold(n_splits=5, shuffle=True, random_state=3)
```

Code 6: K-fold Cross Validation

I split the training dataset into 5 folders by setting the hyper-parameter n_splits as 5, and I set the hyper-parameter shuffle = True to shuffle the dataset.

Model training is based on the fit function from the parent class of GridSearchCV –

BaseSearchCV. Before predicting the topics for the articles in the test dataset, the training data is transformed and visualized to show the distribution of labels in the reduced LDA space.

Finally, the predictions are evaluated using classification_report, which provides metrics include precision, recall, and F1-score.

5.2 Hyper-parameter Tuning

As mentioned in Section 5.1, I leveraged Grid Search to tune the hyper-parameters in the training process, as shown in Code 7.

```
param_grid = {
    "scaler": [StandardScaler(), MinMaxScaler()],
    "lda__solver": ["svd", "lsqr", "eigen"],
    "lda__tol": [1e-4, 1e-3, 1e-2],
    "lda__store_covariance": [True, False],
}
```

Code 7: Hyper-parameters to be tuned

For the standardization hyper-parameters, I conducted training process on both StandardScaler and MinMaxScaler, where StandardScaler standardizes features by removing the mean and scaling to unit variance and MinMaxScaler scales features to a given range, usually between 0 and 1.

Additionally, LDA model hyper-parameters were tuned in the training too.

(1) lda_solver: The algorithm used to compute the LDA solution. Options are svd, lsqr, and eigen.
(2) lda_tol: The tolerance for convergence.
(3) lda_store_covariance: Whether to store the covariance matrix of each class.

These hyperparameters are critical as they affect the scaling of data and the internal computations of the LDA model. The grid search systematically evaluates combinations of these parameters to identify the configuration that yields the best performance on the validation set.

5.3 Environment

The following table (Table 1) shows the version of python and the package dependencies.

Table 1: Software/Package

Software/Package Environment	Version
Python	3.9.18
Visual Studio Code	1.89.1
Pandas	1.3.5
BeautifulSoup	4.12.2
Urllib3	2.0.3
Nltk	3.8.1
Re2	2022.04.01
Textblob	0.15.3
Wordcloud	1.9.3
Matplotlib	3.8.2
Genism	4.3.0
Numpy	1.23.5
Scikit-learn	1.4.2

The following table (Table 2) shows the hardware dependencies.

Table 2: Hardware Dependencies

CPU	AMD Ryzen 7 5800H with Radeon Graphics
GPU	NVIDIA GeForce RTX 3060 Laptop GPU

5.4 Evaluation

The provided evaluation results give insights into the performance of the machine learning model across different classes (Fisher, R. A., 1936). Here's a detailed breakdown:

- Entertainment: Precision = 0.357, Recall = 0.417, F1-score = 0.385, Support = 12
- Environment: Precision = 0.438, Recall = 0.467, F1-score = 0.452, Support = 15
- Medical: Precision = 0.389, Recall = 0.700, F1-score = 0.500, Support = 10
- Politics: Precision = 0.167, Recall = 0.100, F1-score = 0.125, Support = 10
- Sports: Precision = 0.500, Recall = 0.308, F1-score = 0.381, Support = 13
- Technology: Precision = 0.500, Recall = 0.417, F1-score = 0.455, Support = 12
- Accuracy: 0.403
- Macro Average: Precision = 0.392, Recall = 0.401, F1-score = 0.383, Support = 72
- Weighted Average: Precision = 0.401, Recall = 0.403, F1-score = 0.390, Support = 72

A visualized version of the performance is as shown below:

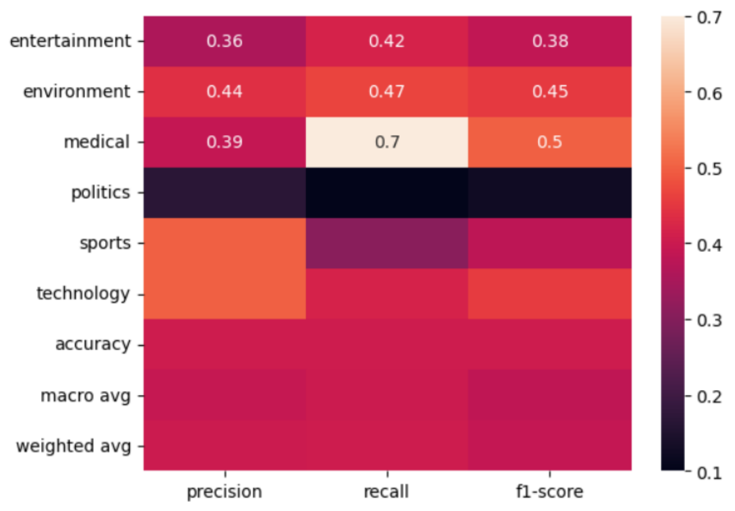


Figure 10: Evaluation Results

A confusion matrix (Figure 11) visualizes the model’s predictions against the actual labels, highlighting where the model makes errors. This visualization helps identify specific misclassifications, such as whether political articles are often misclassified as another category.

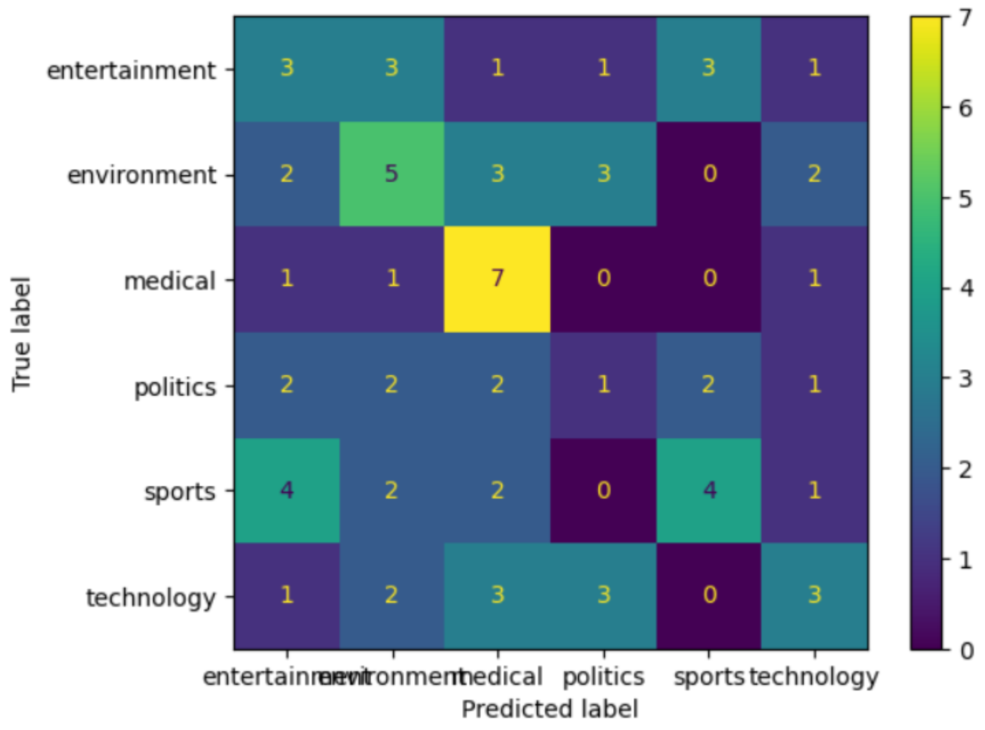


Figure 11: Confusion Matrix

The overall accuracy of the model is 40.28%, which is relatively low, indicating the model's struggles to correctly classify the news articles. The F1-scores for individual classes vary significantly, with the 'Politics' class performing the worst (F1-score = 0.125) and the 'Medical' class performing the best (F1-score = 0.500). The low precision and recall for 'Politics' suggest the model has difficulty distinguishing political articles from other categories. The 'Medical' category, with a higher recall, indicates the model is better at identifying true positives for medical articles but might also include more false positives, as shown by its precision.

A scatter plot of the transformed training labels in LDA space (Figure 11) shows how articles are spread out across different classes. Also, the scatter plot would likely show significant overlap between classes, indicating why the model struggles to classify some categories correctly.

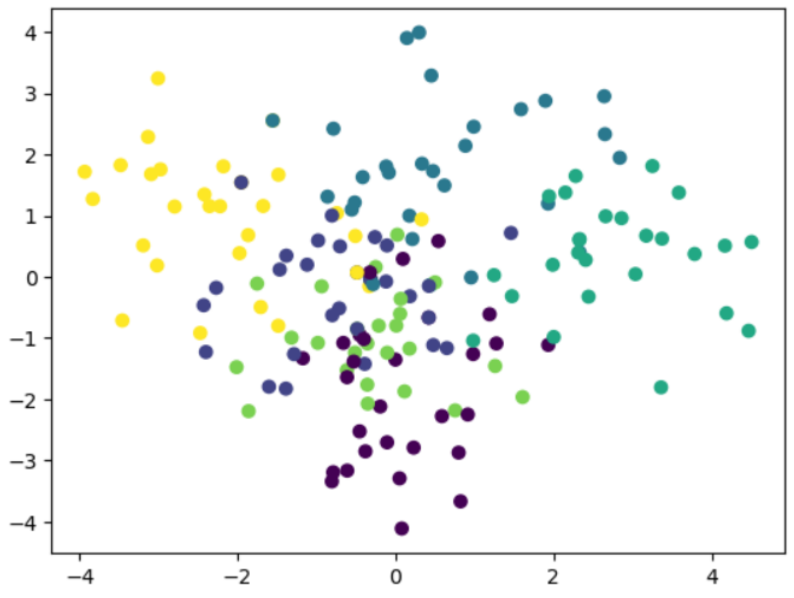


Figure 11: Distribution of the labels

Even though the hyper-parameters have been tuned by the Grid Search algorithm:

Hyper-parameter	Value
lda_solver	svd
lda_store_covariance	False
lda_tol	0.0001
Scaler	StandardScaler()

The model’s performance is still not well in predicting the labels for the data. This may due to feature overlap, similar features across different classes make it difficult for the model to distinguish between them, leading to lower precision and recall for those categories. As seen in the confusion matrix, articles from underrepresented or overlapping classes are more frequently misclassified, demonstrating the impact of sampling biases on the model's performance.

References

Conneau, A., Schwenk, H., Barrault, L., & LeCun, Y. (2017). Very Deep Convolutional Networks for Text Classification. *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 1, Long Papers*, 1107-1116. https://aclanthology.org/E17-1104/

Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, 4171-4186. https://aclanthology.org/N19-1423/

Fisher, R. A. (1936). The use of multiple measurements in taxonomic problems. *Annals of Eugenics, 7*(2), 179-188. https://doi.org/10.1111/j.1469-1809.1936.tb02137.x

Joachims, T. (1998). Text Categorization with Support Vector Machines: Learning with Many Relevant Features. *Proceedings of the 10th European Conference on Machine Learning*, 137-142. https://link.springer.com/chapter/10.1007/BFb0026683

Kohavi, R. (1995). A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI)* (pp. 1137-1145). https://www.ijcai.org/Proceedings/95/Papers/016.pdf

Sebastiani, F. (2002). Machine learning in automated text categorization. *ACM Computing Surveys (CSUR), 34*(1), 1-47. https://doi.org/10.1145/505282.505283

McCallum, A., & Nigam, K. (1998). A Comparison of Event Models for Naive Bayes Text Classification. *AAAI-98 Workshop on Learning for Text Categorization*, 752-758. https://www.aaai.org/Papers/Workshops/1998/WS-98-05/WS98-05-011.pdf

Github Repository

https://github.com/HeliosGuo/News-Topics-Prediction

Code

```
# Crawler

# Import the necessary libs
import os
import time
import requests
import pandas as pd
```

```
from bs4 import BeautifulSoup
from urllib.parse import urljoin

# Global variables
# Create corpus path
folder_path = os.path.join("./corpus")
os.makedirs(folder_path, exist_ok=True)

# Base link
base_link = https://www.voanews.com

# Crawler Class
class my_crawler():
    def __init__(self, start_link, keyword):
        self.link_to_explore = start_link
        self.keyword = keyword
        self.url_prefix, self.url_suffix = start_link.split("pi=1")
        self.articles = pd.DataFrame(columns = ['keyword', 'article'])
        self.pagecount = 1
        self.next = True
        self.hrefs = set()

    def run(self):
        while self.next:
            if self.pagecount > 2:
                self.extract_data_at_once()
                self.save_data_to_file()
                self.next = False
            try:
                self.get_urls()
                self.pagecount = self.pagecount + 1
            except:
                print ("Cannot get the page " + self.link_to_explore)
                self.next = False
                raise

    def get_urls(self):

        # Regex Match
        self.link_to_explore = self.url_prefix + "pi=" + str(self.pagecount) + self.url_suffix

        # Send requests
        response = requests.get(self.link_to_explore)
        html_content = response.content
        # Parse the html contents
        soup = BeautifulSoup(html_content, "html.parser")

        container = soup.find(class_="media-block-wrap")

        # Get all the hrefs (repetition included)
        if container:
            links = container.find_all("a")
            links_exclude = container.find_all("a", class_ = "links__item-link")
            links = list(set(links) - set(links_exclude))
        else:
            print("Container not found!")
            return

        for link in links:
            href = link.get("href")
            self.hrefs.add(href)

    def extract_data_at_once(self):

        for idx, half_link in enumerate(self.hrefs):
            full_link = urljoin(base_link, half_link)

            try:
                response = requests.get(full_link)
                response.raise_for_status()    # Check the abnormal code
                html = BeautifulSoup(response.content, "html.parser")
                # articles = html.find_all(class_="RichTextStoryBody RichTextBody")
                articles = html.find_all("p")
                if articles is None:
                    continue

                # Get the content of the article
                content = ' '.join([article.get_text() for article in articles[1:]])

                # Control the frequency of the requests
                time_between_requests = 6 / 3    # 2 RPM
                time.sleep(time_between_requests)

                # Write down the articles
                self.articles.loc[len(self.articles)] = [self.keyword, content]

                if len(self.articles) % 10 == 0:
```

```
                print("Finish loading", len(self.articles), "aricles...")


            except requests.RequestException as e:
                print(f"WRONG REQUEST{str(e)}")

    def save_data_to_file(self):
        self.articles.to_csv ('./corpus/%s_dataset.csv' % self.keyword, index = None,
header=True, encoding='utf_8_sig')

# Merge the datasets of different topics
def merge_files():

    SaveFile_Path = r"./corpus"
    SaveFile_Name = r"dataset.csv"

    # Save all the file to a list
    file_list = os.listdir(folder_path)

    # Read the first file (header = True)
    df = pd.read_csv(folder_path + "\\" + file_list[0])

    # Save the first file to the merged file
    df.to_csv(SaveFile_Path + "\\" + SaveFile_Name, encoding="utf_8_sig", index=False)

    # Traverse the files and merge the rest of the files
    for i in range(1, len(file_list)):
        df = pd.read_csv(folder_path + "\\" + file_list[i])
        df.to_csv(
            SaveFile_Path + "\\" + SaveFile_Name,
            encoding="utf_8_sig",
            index=False,
            header=False,
            mode="a+",
        )

# Start Crawling
# Six different topics/keywords
keywords = ['medical', 'technology', 'entertainment', 'sports', 'environment', 'politics']
for keyword in keywords:
    url_start = "https://www.voanews.com/s?k=%s&tab=all&pi=1&r=any&pp=20" %
keyword
    try:
        mycrawler = my_crawler(url_start, keyword)
        mycrawler.run()
        print("Finished loading all the articles from topic: %s !" % keyword)
    except:
        raise
# Merge the csv datasets
merge_files()

# Data Clean & Visualization

# Import libs
import nltk
import re
import csv
import string
from nltk.corpus import stopwords
from textblob import TextBlob

# Download necessary files
nltk.download('stopwords')
nltk.download('punkt')
nltk.download('wordnet')

# Function definition
def clean_text(text):
    # Eliminate HTML lables and entities
    text = re.sub(r"<.*?>", "", text)
    # Subtract punctuations, including unicode characters
    # Create a translation table
    translator = str.maketrans(
        {
            "\u2018": "",
            "\u2019": "",    # Single quotes
            "\u201c": "",
            "\u201d": "",    # Double quotes
            "\u2026": "",
            "\u2013": "",
            "\u2014": "",    # Ellipsis and dashes
        }
    )
    translator.update(str.maketrans("", "", string.punctuation))
    text = text.translate(translator)
    # Lowercasing
    text = text.lower()
    # Subtract stop words
    stop_words = stopwords.words("english")
    stop_words.extend(
        [
            "from",
            "said",
            "say",
            "reported",
            "news",
            "according",
            "would",
            "could",
            "posting",
            "article",
            "one",
            "two",
            "three",
        ]
    )
    text = " ".join([word for word in text.split() if word not in stop_words])

    # Correct possible spelling errors
    text = str(TextBlob(text).correct())

    # Detele extra white spaces
    text = re.sub(r"\s+", " ", text).strip()

    return text

# Save the clean data
# Read the data
with open("./corpus/dataset.csv", 'r', encoding='utf_8_sig') as f:
    reader = csv.reader(f)
    data = list(reader)

# Create a new list storing the cleaned data
cleaned_data = ['clean_data']
for idx, text in enumerate(data[1:]):
    cleaned_data.append(clean_text(text[1]))
    if (idx + 1) % 20 == 0:
        print("Finished cleaning", idx+1, "articles!")

# Merge the columns
for idx, row in enumerate(data):
        row.append(cleaned_data[idx])
data = pd.DataFrame(data)

# Save the data
data.to_csv('./corpus/dataset.csv',index=False,mode='w',header=None, encoding='utf_8_sig')

# Visualization

# Import libs
from collections import Counter
from nltk.tokenize import word_tokenize
from wordcloud import WordCloud
import matplotlib.pyplot as plt
from nltk import ngrams

# Function definition
# Tokenize the text
def tokenize(text):
    return word_tokenize(text)

# Word Frequency Analysis: Identify the most frequent words.
def word_frequency(tokens):
    return Counter(tokens)

# N-gram Analysis: Explore common bigrams, trigrams, etc.
def ngram_analysis(tokens, n=2):
    return list(ngrams(tokens, n))

# Visualization: Use word clouds for visual analysis.
def plot_wordcloud(text):
    wordcloud = WordCloud(width=800, height=400).generate(text)
    plt.figure(figsize=(10, 5))
    plt.imshow(wordcloud, interpolation='bilinear')
    plt.axis('off')
    plt.show()

def lemmatize_sentence_textblob(sentence):
    blob = TextBlob(sentence)
    lemmatized_sentence = [word.lemmatize() for word in blob.words]
    return ' '.join(lemmatized_sentence)

def draw_from_dict(dicdata,RANGE, heng=0):
    # dicdata：  data from the dict
    # RANGE：  range of the dict that is going to show
    # heng=0 : vertical bars; heng=1 : horizon bars
    by_value = sorted(dicdata.items(),key = lambda item:item[1],reverse=True)
    x = []
    y = []
    for d in by_value:
        x.append(d[0])
        y.append(d[1])
    if heng == 0:
        plt.bar(x[0:RANGE], y[0:RANGE])
        plt.show()
        return
    elif heng == 1:
        plt.barh(x[0:RANGE], y[0:RANGE])
        plt.show()
        return
    else:
        return "heng has to be 0 or 1!!"

# Read the data
df = pd.read_csv('./corpus/dataset.csv', encoding='utf_8')
print(df.head())
# Merge all the articles into one string
long_string = ' '.join(article for article in df['clean_data'].values)
lemmatized_long_string = lemmatize_sentence_textblob(long_string)

# The visualization before lemmatize

# Single word frequency
word_freq = word_frequency(tokenize(long_string))
print(word_freq)
draw_from_dict(dict(word_freq), 20, heng=1)

# Bi-grams frequency
ngram_ana = ngram_analysis(tokenize(long_string))
bi_grams = []
for item in ngram_ana:
    a = ' '.join(word for word in list(item))
    bi_grams.append(a)
bi_word_freq = word_frequency(bi_grams)
print(bi_word_freq)
draw_from_dict(dict(bi_word_freq), 20, heng=1)
print("Word Cloud for clean data before lemmatize: ")
plot_wordcloud(long_string)
```

```
# The visualization after lemmatize

# Single word frequency
lemmatized_word_freq = word_frequency(tokenize(lemmatized_long_string))
print(lemmatized_word_freq)
draw_from_dict(dict(lemmatized_word_freq), 20, heng=1)

# Bi-grams frequency
lemmatized_ngram_ana = ngram_analysis(tokenize(lemmatized_long_string))
lemmatized_bi_grams = []
for item in lemmatized_ngram_ana:
    a = ' '.join(word for word in list(item))
    lemmatized_bi_grams.append(a)
lemmatized_bi_word_freq = word_frequency(lemmatized_bi_grams)
print(lemmatized_bi_word_freq)
draw_from_dict(dict(lemmatized_bi_word_freq), 20, heng=1)
print("Word Cloud for clean data after lemmatize: ")
plot_wordcloud(lemmatized_long_string)

# Feature Extraction

# Import libs
from gensim.models import Word2Vec
import numpy as np

# Word2Vec features
# We leverage pre-trained model word2vec to generate features
def word2vec_embedding(tokens):
    # Parameters:
    # - sg: 1 for skip-gram; 0 for CBOW, default = 0
    # - vector_size: dimensions of the word vectors
    # - window: context window size
    # - min_count: minimum frequency of words to consider
    model = Word2Vec(tokens, vector_size=100, window=5, min_count=1, workers=4)
    return model

word_emb = word2vec_embedding([tokenize(article) for article in list(df['clean_data'].values)])

print(f"Vocabulary size: {len(word_emb.wv.key_to_index)}")

# Generate article vectors
def get_article_vector(article, model):
    words = tokenize(article)
    word_vectors = []
    for word in words:
        if word in model.wv:
            word_vectors.append(model.wv[word])
        else:
            print(word, "isn't in the vocabulary!")

    if word_vectors:
        article_vector = np.mean(word_vectors, axis=0)
    else:
        # Handle case where no words in sentence are in the vocabulary
        article_vector = np.zeros(model.vector_size)
    return article_vector

# Generate vectors for the entire corpus
article_vectors = [get_article_vector(article, word_emb) for article in
list(df['clean_data'].values)]

print("Article vectors:")
for vec in article_vectors:
    print(vec)

# Model Training & Evaluating

# Import libs
from sklearn.model_selection import train_test_split
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.metrics import classification_report
from sklearn.model_selection import GridSearchCV, KFold
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler, MinMaxScaler

# Example labels for a classification task
labels = list(df["keyword"].values)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(
    article_vectors, labels, test_size=0.3, random_state=3
)

pipeline = Pipeline(
    [
        ("scaler", StandardScaler()),   # Normalize the data
        ("lda", LinearDiscriminantAnalysis()),   # LDA model
    ]
)

# Grid search
param_grid = {
    "scaler": [StandardScaler(), MinMaxScaler()],
    "lda__solver": ["svd", "lsqr", "eigen"],
    "lda__tol": [1e-4, 1e-3, 1e-2],
    "lda__store_covariance": [True, False],
}

# K-fold
# n_splits: Number of folds. Must be at least 2.
# shuffle: if shuffle the dataset
# random_state: When shuffle is True, random_state controls the randomness of each fold.
kfold = KFold(n_splits=5, shuffle=True, random_state=3)

grid_search = GridSearchCV(
    pipeline, param_grid, cv=kfold, scoring="accuracy", n_jobs=-1
)

# Train a classifier
grid_search.fit(X_train, y_train)
for k, v in grid_search.get_params().items():
    print(k, ": ", v)

# best_model = grid_search.best_estimator_
# print(f"Best Model: {best_model}")

# Convert labels to numbers to visualize
labels_to_numbers = []
for idx, item in enumerate(y_train):
    if item == "medical":
        labels_to_numbers.append(0)
    elif item == "technology":
        labels_to_numbers.append(1)
    elif item == "entertainment":
        labels_to_numbers.append(2)
    elif item == "sports":
        labels_to_numbers.append(3)
    elif item == "environment":
        labels_to_numbers.append(4)
    elif item == "politics":
        labels_to_numbers.append(5)
    else:
            continue

# Visualize the distribution of labels
x_new = grid_search.transform(X_train)
plt.scatter(x_new[:, 0], x_new[:, 1], marker="o", c=labels_to_numbers)
plt.show()

# Predict and evaluate
y_pred = grid_search.predict(X_test)
cl_re = classification_report(
    y_test, y_pred, output_dict=True
)
print(cl_re)

# Visualize the results
import seaborn as sns
sns.heatmap(pd.DataFrame(cl_re).iloc[:-1, :].T, annot=True)

from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

# Calculate the confusion matrix
cm = confusion_matrix(y_test, y_pred, labels=['entertainment', 'environment', 'medical',
'politics', 'sports', 'technology'])

# Display the confusion matrix
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=['entertainment',
'environment', 'medical', 'politics', 'sports', 'technology'])
disp.plot()
plt.show()
```