```python

import pandas as pd

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.decomposition import PCA

from sklearn.metrics import silhouette\_score

from sklearn.metrics.pairwise import cosine\_distances

import numpy as np

import matplotlib.pyplot as plt

from nltk.tokenize import word\_tokenize

from nltk.stem import WordNetLemmatizer

from collections import Counter

from wordcloud import WordCloud

import nltk

nltk.download('punkt')

nltk.download('wordnet')

```

- \*\*Imports\*\*: Various libraries are imported for data manipulation (`pandas`, `numpy`), text processing (`nltk`), machine learning (`sklearn`), visualization (`matplotlib`), and generating word clouds (`wordcloud`).

- \*\*nltk.download('punkt')\*\* and \*\*nltk.download('wordnet')\*\*: Downloads required NLTK datasets for tokenization and lemmatization.

```python

# Reading the csv file

df = pd.read\_csv("cleanednewsarticles.csv")

# Convert the date column to datetime format (dd-mm-yyyy hh:mm)

df['date'] = pd.to\_datetime(df['date'], format='%d-%m-%Y %H:%M')

```

- \*\*Reading the csv file\*\*: Loads the dataset into a DataFrame using `pandas`.

- \*\*Convert the date column\*\*: Converts the `date` column to `datetime` format, which allows for easier date manipulation and filtering.

```python

# Function to filter articles by date range

def filter\_by\_date(df, start\_date, end\_date):

mask = (df['date'] >= start\_date) & (df['date'] <= end\_date)

return df[mask]

# User input for date range

start\_date = pd.to\_datetime(input("Enter the start date (dd-mm-yyyy): "), format='%d-%m-%Y')

end\_date = pd.to\_datetime(input("Enter the end date (dd-mm-yyyy): "), format='%d-%m-%Y')

# Filter articles by the specified date range

df = filter\_by\_date(df, start\_date, end\_date)

```

- \*\*filter\_by\_date function\*\*: Filters the DataFrame to include only rows within the specified date range.

- \*\*User input for date range\*\*: Prompts the user to input the start and end dates, which are then converted to `datetime` format.

- \*\*Filter articles\*\*: Filters the dataset based on the specified date range using the `filter\_by\_date` function.

```python

# Reading the stopword.txt file

with open('stop\_words.txt', 'r', encoding='utf-8') as file:

stop\_words = set(file.read().splitlines())

```

- \*\*Reading stop\_words.txt\*\*: Loads a list of stop words from a file into a Python set. Stop words are common words (e.g., "and", "the") that are often removed in text preprocessing to focus on more meaningful words.

```python

# Removing the stopwords

df['cleaned\_article'] = df['cleaned\_article'].apply(lambda text: ' '.join(

[word for word in word\_tokenize(text) if word not in stop\_words]))

```

- \*\*Removing the stopwords\*\*: Tokenizes each article and removes any stop words, then joins the remaining words back into a single string.

```python

# Lemmatization

lemmatizer = WordNetLemmatizer()

df['cleaned\_article'] = df['cleaned\_article'].apply(lambda text: ' '.join(

[lemmatizer.lemmatize(word) for word in text.split()]))

```

- \*\*Lemmatization\*\*: Reduces words to their base form (e.g., "running" to "run") using the WordNet lemmatizer.

```python

# TF-IDF Vectorization

vectorizer = TfidfVectorizer(max\_features=1000)

X = vectorizer.fit\_transform(df['cleaned\_article']).toarray()

```

- \*\*TF-IDF Vectorization\*\*: Converts the cleaned text into a numerical matrix using the TF-IDF method, which assigns a weight to each word based on its importance in the document and across the entire dataset. The `max\_features=1000` parameter limits the matrix to the top 1000 features (words).

```python

# Finding the best number of clusters for the value of K using the elbow method and silhouette score

wcss = []

sil\_scores = []

K = range(2, 11) # Start from 2 as silhouette score is not defined for 1 cluster

```

- \*\*WCSS and Silhouette Score Arrays\*\*: Initializes lists to store Within-Cluster Sum of Squares (WCSS) and silhouette scores for different values of K (number of clusters).

- \*\*K range\*\*: Defines a range of K values to test, starting from 2 (since silhouette score is not defined for a single cluster).

```python

for k in K:

np.random.seed(42)

initial\_centroids = X[np.random.choice(X.shape[0], k, replace=False)]

for \_ in range(100):

distances = cosine\_distances(X, initial\_centroids)

labels = np.argmin(distances, axis=1)

new\_centroids = np.array([X[labels == j].mean(axis=0) for j in range(k)])

if np.all(initial\_centroids == new\_centroids):

break

initial\_centroids = new\_centroids

wcss.append(np.sum(np.min(distances, axis=1)\*\*2))

sil\_scores.append(silhouette\_score(X, labels, metric='cosine'))

```

- \*\*Initial Centroids\*\*: Randomly selects initial centroids for the clusters.

- \*\*Distance Calculation\*\*: Computes cosine distances between each data point and the centroids.

- \*\*Assign Labels\*\*: Assigns each data point to the nearest centroid.

- \*\*Update Centroids\*\*: Calculates new centroids as the mean of all points assigned to each cluster. The loop continues until the centroids do not change.

- \*\*WCSS Calculation\*\*: Computes the WCSS for the current number of clusters and adds it to the list.

- \*\*Silhouette Score Calculation\*\*: Computes the silhouette score for the current number of clusters and adds it to the list.

```python

# Plotting the elbow method

plt.figure(figsize=(8, 5))

plt.plot(K, wcss, 'bo-')

plt.xlabel('Number of clusters (K)')

plt.ylabel('Within-Cluster Sum of Squares (WCSS)')

plt.title('Elbow Method for Optimal K')

plt.show()

# Plotting the silhouette scores

plt.figure(figsize=(8, 5))

plt.plot(K, sil\_scores, 'bo-')

plt.xlabel('Number of clusters (K)')

plt.ylabel('Silhouette Score')

plt.title('Silhouette Score for Optimal K')

plt.show()

```

- \*\*Elbow Method Plot\*\*: Plots the WCSS values for different K values to visualize the "elbow" point where adding more clusters no longer significantly reduces the WCSS.

- \*\*Silhouette Score Plot\*\*: Plots the silhouette scores for different K values to identify the optimal number of clusters.

```python

# Best value of K

best\_k = K[np.argmax(sil\_scores)]

print(f'Best number of clusters (K) based on silhouette score: {best\_k}')

```

- \*\*Best Value of K\*\*: Identifies the optimal number of clusters as the K value with the highest silhouette score.

```python

# Using PCA to reduce the dimension

pca = PCA(n\_components=2)

pca\_components = pca.fit\_transform(X)

# Adding PCA components to dataframe

df['x'] = pca\_components[:, 0]

df['y'] = pca\_components[:, 1]

```

- \*\*PCA for Dimensionality Reduction\*\*: Uses Principal Component Analysis (PCA) to reduce the dimensionality of the TF-IDF matrix to 2 dimensions for visualization.

- \*\*Adding PCA Components\*\*: Adds the PCA components to the DataFrame.

```python

# Initializing K means Clustering with the best K

np.random.seed(42)

initial\_centroids = X[np.random.choice(X.shape[0], best\_k, replace=False)]

# Using Cosine Similarity

for \_ in range(100):

distances = cosine\_distances(X, initial\_centroids)

labels\_cosine = np.argmin(distances, axis=1)

new\_centroids = np.array([X[labels\_cosine == j].mean(axis=0) for j in range(best\_k)])

if np.all(initial\_centroids == new\_centroids):

break

initial\_centroids = new\_centroids

# Adding Cluster labels

df['cosine\_cluster'] = labels\_cosine

```

- \*\*K-Means Clustering\*\*: Initializes centroids and performs K-means clustering using cosine similarity to assign cluster labels.

- \*\*Adding Cluster Labels\*\*: Adds the cluster labels to the DataFrame.

```python

# Word Cloud for each cluster

for cluster in range(best\_k):

cluster\_articles = df[df['cosine\_cluster'] == cluster]

combined\_text = ' '.join(cluster\_articles['cleaned\_article'])

word\_counts = Counter(combined\_text.split())

wordcloud = WordCloud(width=800, height=400, background\_color='white').generate\_from\_frequencies(word\_counts)

plt.figure(figsize=(10, 6))

plt.imshow(wordcloud, interpolation='bilinear')

plt.title(f'Word Cloud for Cluster {cluster}')

plt.axis('off')

plt.show()

```

- \*\*Word Cloud for Each Cluster\*\*: Generates and displays a word cloud for each cluster, showing the most frequent words.

- \*\*Word Counts\*\*: Counts the frequency of each word in the cluster's articles

.

- \*\*Generate Word Cloud\*\*: Creates a word cloud based on the word frequencies and displays it.

```python

# Distribution of Data Points in Clusters

plt.figure(figsize=(8, 5))

plt.hist(labels\_cosine, bins=range(best\_k+1), align='left', color='blue', edgecolor='black')

plt.title('Distribution of Data Points in Clusters (Cosine Similarity)')

plt.xlabel('Cluster')

plt.ylabel('Number of Data Points')

plt.xticks(range(best\_k))

plt.grid(True)

plt.show()

```

- \*\*Distribution of Data Points\*\*: Plots a histogram showing the distribution of data points across the clusters.

```python

# Silhouette score for Cosine Similarity

sil\_score\_cosine = silhouette\_score(X, labels\_cosine, metric='cosine')

print(f'Silhouette Score (Cosine): {sil\_score\_cosine}')

```

- \*\*Silhouette Score for Cosine Similarity\*\*: Computes and prints the silhouette score for the final clustering using cosine similarity. This score measures the quality of the clustering, with higher values indicating better-defined clusters.