#pip install pypandoc-binary

#pip install nltk

#pip install sumy

import os

import pypandoc

folderpath = ".\Files"

# Get a list of all files in the folderpath

entries = os.listdir(folderpath)

# Iterate through each file and convert to plain text

for entry in entries:

if entry.lower().endswith(".docx"):

try:

output\_file = entry.lower().replace(".docx", ".txt")

output\_path = os.path.join('./Files\_Text', output\_file)

input\_file = os.path.join(folderpath, entry.lower())

pypandoc.convert\_file(input\_file, 'plain', format='docx', outputfile=output\_path)

print(f"{output\_file} is written")

except Exception as e:

print(f"Error occurred while converting {entry}: {str(e)}")

import nltk

from nltk.corpus import PlaintextCorpusReader

corpus\_root = './Files\_Text'

newcorpus = PlaintextCorpusReader(corpus\_root, '.\*')

fileids=newcorpus.fileids()

fileids

import pandas as pd

colnames=['Document','Body','Title']

df=pd.DataFrame(columns=colnames)

count=1

lst=[0,'','']

for j in range(len(fileids)):

print(fileids[j]+" is getting read...")

x=newcorpus.open(fileids[j]).read().split('\r\n\r\n')

startindexes = []

endindexes = []

for i, line in enumerate(x):

if line == 'Body':

startindexes.append(i)

if 'Load-Date:' in line:

endindexes.append(i)

if len(startindexes) == 0 or len(endindexes) == 0:

print("No 'BODY' or 'LOAD-DATE:' found in the text file.")

continue # Skip processing this file

cnt=1

titles=[]

for k,s in enumerate(x):

if str(cnt)+"." in s:

titles.append(s)

cnt+=1

print("Length of titles in ",fileids[j], "are: ",len(titles)-1)

for ind in range(len(startindexes)):

if ind >= len(endindexes):

print("No corresponding 'LOAD-DATE:' found for 'BODY' index:", ind)

continue # Skip this iteration

lst[0] = count

lst[1] = ''.join(x[startindexes[ind] + 2 : endindexes[ind]])

lst[2] = titles[ind].strip()

df.loc[len(df.index)] = lst

count += 1

df.shape

df

import matplotlib.pyplot as plt

import seaborn as sns

# Calculate article lengths using apply()

df['Article Length'] = df['Body'].apply(len)

# Create a violin plot using Seaborn

sns.violinplot(y=df['Article Length'])

# Set plot title and labels

plt.title('Length of Article(Character)')

plt.xlabel('Articles')

plt.ylabel('Length')

# Display the plot

plt.show()

import numpy as np

# Calculate the length of the body text for each article

df['Body\_length'] = df['Body'].str.len()

# Convert the body length values to numeric

df['Body\_length'] = pd.to\_numeric(df['Body\_length'], errors='coerce')

# Assuming your DataFrame has a column named 'body' containing the numerical variable

Q1 = df['Body\_length'].quantile(0.25)

Q3 = df['Body\_length'].quantile(0.75)

IQR = Q3 - Q1

# Define the lower and upper bounds for outliers

lower\_bound = Q1 - 1.5 \* IQR

upper\_bound = Q3 + 1.5 \* IQR

# Remove outliers from the DataFrame based on the body length

filtered\_df = df[(df['Body\_length'].notnull()) & (df['Body\_length'] >= lower\_bound) & (df['Body\_length'] <= upper\_bound)]

# Reset the index of the filtered DataFrame

filtered\_df1 = filtered\_df.reset\_index(drop=True)

# Remove rows with NaN values from the DataFrame

filtered\_df1.dropna(inplace=True)

filtered\_df1 = filtered\_df1[filtered\_df1['Body'].str.len() > 0]

filtered\_df1

# Create a violin plot using Seaborn

sns.violinplot(y=filtered\_df1['Body\_length'])

# Set plot title and labels

plt.title('Length of Article(Character)')

plt.xlabel('Articles')

plt.ylabel('Length')

# Display the plot

plt.show()

# Calculate average article length

average\_length = filtered\_df1['Body\_length'].mean()

# Print the average length

print("Average article length:", average\_length)

import string

# Define a function to remove punctuation from a text

def remove\_punctuation(text):

no\_punct = ''.join([c for c in text if c not in string.punctuation])

return no\_punct

# Apply the remove\_punctuation function to the 'Body' column

df['Body Without Punctuation'] = filtered\_df1['Body'].apply(remove\_punctuation)

# Print the modified DataFrame

print(df['Body Without Punctuation'])

# Remove NaN values from the 'Body Without Punctuation' column

df.dropna(subset=['Body Without Punctuation'], inplace=True)

filtered\_df1 = filtered\_df1[df['Body Without Punctuation'].str.len() > 0]

filtered\_df1

import nltk

from nltk.corpus import stopwords

from nltk.tokenize import word\_tokenize

nltk.download('punkt')

nltk.download('stopwords')

import string

# Retrieve the list of stopwords

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

# Define a function to remove stopwords from a text

def remove\_stopwords(text):

tokens = word\_tokenize(text)

filtered\_tokens = [token for token in tokens if token.lower() not in stop\_words]

no\_stopwords\_text = ' '.join(filtered\_tokens)

return no\_stopwords\_text

# Apply the remove\_stopwords function to the 'Body' column

df['Body Without Stopwords'] = df['Body Without Punctuation'].apply(remove\_stopwords)

# Print the modified DataFrame

print(df['Body Without Stopwords'])

# Define a function to calculate the frequency of non-stopwords in a text

def calculate\_non\_stopword\_frequency(text):

tokens = word\_tokenize(text)

non\_stopwords = [token for token in tokens if token.lower() not in stop\_words]

non\_stopwords\_count = len(non\_stopwords)

return non\_stopwords\_count

# Apply the calculate\_non\_stopword\_frequency function to the 'Body' column

df['Non-Stopword Frequency'] = df['Body Without Punctuation'].apply(calculate\_non\_stopword\_frequency)

# Print the modified DataFrame

print(df['Non-Stopword Frequency'])

# Define a function to calculate the frequency of non-stopwords in a text

def calculate\_non\_stopword\_frequency(text):

tokens = word\_tokenize(text)

non\_stopwords = [token for token in tokens if token.lower() not in stop\_words]

non\_stopwords\_count = len(non\_stopwords)

return non\_stopwords\_count

# Apply the calculate\_non\_stopword\_frequency function to the 'Body' column

df['Non-Stopword Frequency'] = df['Body Without Punctuation'].apply(calculate\_non\_stopword\_frequency)

# Print the modified DataFrame

print(df['Non-Stopword Frequency'])

# Define a function to calculate the frequency of stopwords in a text

def calculate\_stopword\_frequency(text):

tokens = word\_tokenize(text)

stopwords\_count = sum(1 for token in tokens if token.lower() in stop\_words)

return stopwords\_count

# Apply the calculate\_stopword\_frequency function to the 'Body' column

df['Stopword Frequency'] = df['Body Without Punctuation'].apply(calculate\_stopword\_frequency)

# Print the modified DataFrame

print(df['Stopword Frequency'])

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

sns.violinplot(data=df, y=df['Stopword Frequency'])

plt.xlabel('Stopword Frequency')

plt.ylabel('Frequency')

plt.title('Distribution of Stopword Frequency')

plt.show()

# Calculate average article length

average\_frquency = df['Stopword Frequency'].mean()

# Print the average length

print("Average frequency count of stopwords::", average\_frquency)

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

sns.violinplot(data=df, y=df['Non-Stopword Frequency'])

plt.xlabel('Non-Stopword Frequency')

plt.ylabel('Frequency')

plt.title('Distribution of Not-Stopword Frequency')

plt.show()

# Calculate average article length

average\_frquency = df['Non-Stopword Frequency'].mean()

# Print the average length

print("Average frequency count of Non-stopwords::", average\_frquency)

import nltk

from sklearn.feature\_extraction.text import CountVectorizer

nltk.download('stopwords')

from nltk.corpus import stopwords

count\_vectorizer = CountVectorizer(stop\_words=stopwords.words('english'), min\_df=5, max\_df=0.7)

count\_vectors = count\_vectorizer.fit\_transform(filtered\_df1['Body'])

from sklearn.decomposition import LatentDirichletAllocation

lda\_model = LatentDirichletAllocation(n\_components = 10, random\_state=42)

W\_lda\_matrix = lda\_model.fit\_transform(count\_vectors)

H\_lda\_matrix = lda\_model.components\_

from sklearn.feature\_extraction.text import TfidfVectorizer

tfidf\_text = TfidfVectorizer(stop\_words=stopwords.words('english'), min\_df=5, max\_df=0.7)

vectors\_text = tfidf\_text.fit\_transform(filtered\_df1['Body'])

vectors\_text.shape

def display\_topics(model, features, no\_top\_words=10):

for topic, word\_vector in enumerate(model.components\_):

total = word\_vector.sum()

largest = word\_vector.argsort()[::-1] # invert sort order

print("\nTopic %02d" % topic)

for i in range(0, no\_top\_words):

print(" %s (%2.2f)" % (features[largest[i]],word\_vector[largest[i]]\*100.0/total))

display\_topics(lda\_model, tfidf\_text.get\_feature\_names\_out())

import pyLDAvis.lda\_model

lda\_display = pyLDAvis.lda\_model.prepare(lda\_model, count\_vectors,count\_vectorizer, sort\_topics=False)

pyLDAvis.display(lda\_display)

from sumy.parsers.plaintext import PlaintextParser

from sumy.nlp.tokenizers import Tokenizer

from sumy.summarizers.lex\_rank import LexRankSummarizer

# Initialize the summarizer

summarizer = LexRankSummarizer()

# Function to generate the summary using the specified summarizer

def summarize\_text(text, summarizer):

parser = PlaintextParser.from\_string(text, Tokenizer("english"))

summary = summarizer(parser.document, 1) # You can specify the number of sentences in the summary

return " ".join(str(sentence) for sentence in summary)

# Generate summaries for each article in the 'Body' column

df['Summary'] = filtered\_df1['Body'].apply(lambda x: summarize\_text(x, summarizer))

# Print the summaries

for index, row in df.iterrows():

print(f"Article {index+1} Summary:")

print(row['Summary'])

print()

import nltk

from sklearn.feature\_extraction.text import CountVectorizer

nltk.download('stopwords')

from nltk.corpus import stopwords

count\_vectorizer = CountVectorizer(stop\_words=stopwords.words('english'), min\_df=5, max\_df=0.7)

count\_vectors = count\_vectorizer.fit\_transform(df['Summary'])

from sklearn.decomposition import LatentDirichletAllocation

lda\_model = LatentDirichletAllocation(n\_components = 10, random\_state=42)

W\_lda\_matrix = lda\_model.fit\_transform(count\_vectors)

H\_lda\_matrix = lda\_model.components\_

from sklearn.feature\_extraction.text import TfidfVectorizer

tfidf\_text = TfidfVectorizer(stop\_words=stopwords.words('english'), min\_df=5, max\_df=0.7)

vectors\_text = tfidf\_text.fit\_transform(df['Summary'])

vectors\_text.shape

def display\_topics(model, features, no\_top\_words=10):

for topic, word\_vector in enumerate(model.components\_):

total = word\_vector.sum()

largest = word\_vector.argsort()[::-1] # invert sort order

print("\nTopic %02d" % topic)

for i in range(min(no\_top\_words, len(features))):

if largest[i] < len(features):

print(" %s (%2.2f)" % (features[largest[i]], word\_vector[largest[i]] \* 100.0 / total))

display\_topics(lda\_model, tfidf\_text.get\_feature\_names\_out())

import pyLDAvis.lda\_model

lda\_display = pyLDAvis.lda\_model.prepare(lda\_model, count\_vectors,count\_vectorizer, sort\_topics=False)

pyLDAvis.display(lda\_display)

from gensim import corpora

from gensim import corpora

from gensim.models import LdaModel

from gensim.corpora import Dictionary

from gensim.models import CoherenceModel

# Create a list of documents from the 'Body' column

documents\_body = filtered\_df1['Body'].tolist()

# Create a list of documents from the 'Summary' column

documents\_summary = df['Summary'].tolist()

# Combine the 'Body' and 'Summary' documents

documents = documents\_body + documents\_summary

# Tokenize the documents

tokenized\_documents = [doc.split() for doc in documents]

# Create a dictionary from the tokenized documents

dictionary = corpora.Dictionary(tokenized\_documents)

# Create the bag-of-words representation for 'Body'

bow\_corpus\_body = [dictionary.doc2bow(doc.split()) for doc in documents\_body]

# Create the bag-of-words representation for 'Summary'

bow\_corpus\_summary = [dictionary.doc2bow(doc.split()) for doc in documents\_summary]

num\_topics = 10

# Create the LDA model for 'Body' using the bag-of-words representation

lda\_model\_body = LdaModel(bow\_corpus\_body, num\_topics=num\_topics, id2word=dictionary)

# Create the LDA model for 'Summary' using the bag-of-words representation

lda\_model\_summary = LdaModel(bow\_corpus\_summary, num\_topics=num\_topics, id2word=dictionary)

# Calculate the coherence score for 'Body'

coherence\_model\_body = CoherenceModel(model=lda\_model\_body, texts=tokenized\_documents, dictionary=dictionary, coherence='c\_v')

coherence\_score\_body = coherence\_model\_body.get\_coherence()

# Calculate the coherence score for 'Summary'

coherence\_model\_summary = CoherenceModel(model=lda\_model\_summary, texts=tokenized\_documents, dictionary=dictionary, coherence='c\_v')

coherence\_score\_summary = coherence\_model\_summary.get\_coherence()

print("Coherence Score - Body:", coherence\_score\_body)

print("Coherence Score - Summary:", coherence\_score\_summary)

import matplotlib.pyplot as plt

import numpy as np

# Define the range of topics to evaluate

start\_topic = 2

end\_topic = 10

# Initialize lists to store coherence scores

coherence\_scores\_body = []

coherence\_scores\_summary = []

# Define the number of topics

num\_topics = end\_topic - start\_topic + 1

# Iterate over the range of topics

for num\_topics in range(start\_topic, end\_topic + 1):

# Create the LDA model for 'Body' using the bag-of-words representation

lda\_model\_body = LdaModel(bow\_corpus\_body, num\_topics=num\_topics, id2word=dictionary)

# Create the LDA model for 'Summary' using the bag-of-words representation

lda\_model\_summary = LdaModel(bow\_corpus\_summary, num\_topics=num\_topics, id2word=dictionary)

# Calculate the coherence score for 'Body'

coherence\_model\_body = CoherenceModel(model=lda\_model\_body, texts=tokenized\_documents, dictionary=dictionary, coherence='c\_v')

coherence\_score\_body = coherence\_model\_body.get\_coherence()

# Calculate the coherence score for 'Summary'

coherence\_model\_summary = CoherenceModel(model=lda\_model\_summary, texts=tokenized\_documents, dictionary=dictionary, coherence='c\_v')

coherence\_score\_summary = coherence\_model\_summary.get\_coherence()

# Append the coherence scores to the respective lists

coherence\_scores\_body

coherence\_score\_summary

from gensim import corpora

from gensim.models import LdaModel

from gensim.corpora import Dictionary

from gensim.models import CoherenceModel

# Create a list of documents from the 'Body' column

documents\_body = filtered\_df1['Body'].tolist()

# Create a list of documents from the 'Summary' column

documents\_summary = df['Summary'].tolist()

# Combine the 'Body' and 'Summary' documents

documents = documents\_body + documents\_summary

# Tokenize the documents

tokenized\_documents = [doc.split() for doc in documents]

# Create a dictionary from the tokenized documents

dictionary = corpora.Dictionary(tokenized\_documents)

# Create the bag-of-words representation for 'Body'

bow\_corpus\_body = [dictionary.doc2bow(doc.split()) for doc in documents\_body]

# Create the bag-of-words representation for 'Summary'

bow\_corpus\_summary = [dictionary.doc2bow(doc.split()) for doc in documents\_summary]

num\_topics = 10

# Create the LDA model for 'Body' using bag-of-words representation

lda\_model\_body = LdaModel(bow\_corpus\_body, num\_topics=num\_topics, id2word=dictionary)

# Create the LDA model for 'Summary' using bag-of-words representation

lda\_model\_summary = LdaModel(bow\_corpus\_summary, num\_topics=num\_topics, id2word=dictionary)

# Calculate the coherence score for 'Body'

coherence\_model\_body = CoherenceModel(model=lda\_model\_body, texts=tokenized\_documents, dictionary=dictionary, coherence='c\_v')

coherence\_score\_body(TF-IDF) = coherence\_model\_body.get\_coherence()

# Calculate the coherence score for 'Summary'

coherence\_model\_summary = CoherenceModel(model=lda\_model\_summary, texts=tokenized\_documents, dictionary=dictionary, coherence='c\_v')

coherence\_score\_summary(TF-IDF) = coherence\_model\_summary.get\_coherence()

print("Coherence Score - Body:", coherence\_score\_body(TF-IDF))

print("Coherence Score - Summary:", coherence\_score\_summary(TF-IDF))

import matplotlib.pyplot as plt

import numpy as np

# Define the range of topics to evaluate

start\_topic = 2

end\_topic = 10

# Initialize lists to store coherence scores

coherence\_scores\_body = []

coherence\_scores\_summary = []

# Define the number of topics

num\_topics = end\_topic - start\_topic + 1

# Iterate over the range of topics

for num\_topics in range(start\_topic, end\_topic + 1):

# Create the LDA model for 'Body' using bag-of-words representation

lda\_model\_body = LdaModel(bow\_corpus\_body, num\_topics=num\_topics, id2word=dictionary)

# Create the LDA model for 'Summary' using bag-of-words representation

lda\_model\_summary = LdaModel(bow\_corpus\_summary, num\_topics=num\_topics, id2word=dictionary)

# Calculate the coherence score for 'Body'

coherence\_model\_body = CoherenceModel(model=lda\_model\_body, texts=tokenized\_documents, dictionary=dictionary, coherence='c\_v')

coherence\_score\_body(BoW) = coherence\_model\_body.get\_coherence()

# Calculate the coherence score for 'Summary'

coherence\_model\_summary = CoherenceModel(model=lda\_model\_summary, texts=tokenized\_documents, dictionary=dictionary, coherence='c\_v')

coherence\_score\_summary(BoW) = coherence\_model\_summary.get\_coherence()

# Append the coherence scores to the respective lists

coherence\_scores\_body.append(coherence\_score\_body(BoW))

coherence\_scores\_summary.append(coherence\_score\_summary(BoW))

# Plot the coherence scores

x = np.arange(start\_topic, end\_topic + 1)

plt.plot(x, coherence\_scores\_body(TF-IDF), marker='o', label='TFIDF Article')

plt.plot(x, coherence\_scores\_summary(TF-IDF), marker='o', label='TFIDF Summary')

plt.plot(x, coherence\_scores\_body(BoW), marker='o', label='BoW Article')

plt.plot(x, coherence\_scores\_summary(BoW), marker='o', label='BoW Summary')

# Customize the plot

plt.xlabel('Number of Topics')

plt.ylabel('Coherence Score')

plt.title('Coherence Score vs Number of Topics')

plt.legend()

# Display the plot

d= plt.show()

import nltk

nltk.download(["names","punkt","averaged\_perceptron\_tagger","maxent\_ne\_chunker","words","vader\_lexicon","stopwords","shakespeare"])

from nltk.sentiment import SentimentIntensityAnalyzer

from tqdm.notebook import tqdm

sia = SentimentIntensityAnalyzer()

res = {}

for i, row in tqdm(df.iterrows(), total=len(filtered\_df1)):

text = row['Body']

myid = row['Document']

res[myid] = sia.polarity\_scores(text)

vaders = pd.DataFrame(res).T

vaders = vaders.reset\_index().rename(columns={'index': 'Document'})

vaders = vaders.merge(filtered\_df1, how='left')

vaders.head()

import matplotlib.pyplot as plt

import seaborn as sns

ax = sns.lineplot(data=vaders, x='Document', y='compound')

ax.set\_title('Compound Score by Paragraphs')

plt.show()

fig = plt.figure(figsize=(12, 3))

ax = fig.add\_subplot(111)

sns.lineplot(data=vaders, x='Document', y='pos', ax=ax, color='green', label='Positive')

sns.lineplot(data=vaders, x='Document', y='neu', ax=ax, color='blue', label='Neutral')

sns.lineplot(data=vaders, x='Document', y='neg', ax=ax, color='red', label='Negative')

ax.set\_title('Sentiment Scores for Articles')

ax.set\_xlabel('Article Sequence')

ax.set\_ylabel('Polarity Score')

plt.legend()

plt.tight\_layout()

plt.show()

import seaborn as sns

import matplotlib.pyplot as plt

fig = plt.figure(figsize=(12, 6))

ax = fig.add\_subplot(111)

colors = ['green', 'red', 'purple']

sns.violinplot(data=vaders[['pos', 'neu', 'neg']], orient='h', inner="stick",palette=colors)

ax.set\_title('Sentiment Scores for Articles')

ax.set\_xlabel('Polarity Score')

ax.set\_ylabel('Sentiment Category')

plt.tight\_layout()

plt.show()

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.cluster import AgglomerativeClustering

vectorizer = TfidfVectorizer()

X = vectorizer.fit\_transform(filtered\_df1['Title'])

n\_clusters = 3 # Specify the desired number of clusters

clustering = AgglomerativeClustering(n\_clusters=n\_clusters, linkage='ward', affinity='euclidean')

clustering.fit(X.toarray()) # Convert X to an array for compatibility

cluster\_labels = clustering.labels\_

from scipy.cluster.hierarchy import dendrogram, linkage

import matplotlib.pyplot as plt

linkage\_matrix = linkage(X.toarray(), method='ward', metric='euclidean')

dendrogram(linkage\_matrix, labels=filtered\_df1['Title'].values, orientation='top', leaf\_font\_size=8)

plt.xlabel('Document')

plt.ylabel('Complete Linkage Score')

plt.title('Dendrogram')

plt.show()

from sklearn.metrics import pairwise\_distances

from scipy.cluster.hierarchy import linkage

distances = pairwise\_distances(X.toarray(), metric='euclidean')

linkage\_matrix = linkage(distances, method='complete')

complete\_linkage\_score = linkage\_matrix[-1, 2]

complete\_linkage\_score