tes_ody 2_FIX

December 7, 2024

1 Mochammad Daffa Putra Karyudi

2 Import libraries

[1]:
```python
# Standard library imports
import os
import re
import pickle
import warnings

# Data manipulation and numerical computations
import numpy as np
import pandas as pd

# Data visualization
import plotly.express as px
import plotly.graph_objs as go
from plotly.subplots import make_subplots

# Machine Learning and Text Processing
from sklearn.manifold import TSNE
from sklearn.preprocessing import normalize
from sklearn.datasets import fetch_20newsgroups
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.decomposition import LatentDirichletAllocation
from sklearn.feature_extraction.text import TfidfVectorizer

# Natural Language Processing - NLTK
import nltk
from nltk.corpus import stopwords
from nltk.chunk import tree2conlltags
from nltk.stem import WordNetLemmatizer
from nltk.chunk.named_entity import Maxent_NE_Chunker
from nltk.tokenize import word_tokenize, sent_tokenize

# Topic Modeling and Visualization
import pyLDAvis
```

```python
import pyLDAvis.lda_model

# Additional NLP Tools
import spacy
from textblob import TextBlob
from wordcloud import WordCloud

# Deep Learning - Transformers
import torch
import transformers
from transformers import GPT2LMHeadModel, GPT2Tokenizer


# Suppress warnings
warnings.filterwarnings('ignore')

# Set device to CPU
device = torch.device('cpu')
```

[2]:
```python
nltk.download('wordnet')
nltk.download('punkt')
nltk.download('stopwords')
nltk.download('maxent_ne_chunker')
nltk.download('words')
```

```
[nltk_data] Downloading package wordnet to /home/keycode/nltk_data…
[nltk_data]   Package wordnet is already up-to-date!
[nltk_data] Downloading package punkt to /home/keycode/nltk_data…
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package stopwords to
[nltk_data]     /home/keycode/nltk_data…
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package maxent_ne_chunker to
[nltk_data]     /home/keycode/nltk_data…
[nltk_data]   Unzipping chunkers/maxent_ne_chunker.zip.
[nltk_data] Downloading package words to /home/keycode/nltk_data…
[nltk_data]   Package words is already up-to-date!
```

[2]: True

3 Level 1

3.1 Load the dataset

[3]:
```
# Load the dataset
train_data = fetch_20newsgroups(subset='train', remove=('headers', 'footers',␣
↪'quotes'))
```

3.2 Data Cleansing

[4]:
```
def clean_text(text):
    """
    Cleans and preprocesses the input text by performing the following␣
↪operations:

    1. Converts text to lowercase.
    2. Removes URLs.
    3. Removes email addresses.
    4. Eliminates special characters and digits.
    5. Removes words containing 'maxa' or repetitive patterns.
    6. Tokenizes the text using PunktWordTokenizer.
    7. Removes English stopwords.
    8. Lemmatizes the tokens.

    Parameters:
        text (str): The raw input text to be cleaned.

    Returns:
        str: The cleaned and preprocessed text.
    """
    # Initialize lemmatizer and tokenizer
    lemmatizer = WordNetLemmatizer()

    # Convert to lowercase
    text = text.lower()

    # Remove URLs
    text = re.sub(r'http\S+|www\S+|https\S+', '', text, flags=re.MULTILINE)

    # Remove email addresses
    text = re.sub(r'\S+@\S+', '', text)

    # Remove special characters and numbers
    text = re.sub(r'[^a-zA-Z\s]', '', text)

    # Remove repeated characters (more than 2)
    text = re.sub(r'(.)\1{2,}', r'\1', text)
```

```python
    # Remove any word containing 'max' or repetitive patterns
    text = re.sub(r'\b\w*max\w*\b', '', text)

    # Tokenize the text using PunktSentenceTokenizer
    tokens = word_tokenize(text)

    # Remove very short words, lemmatize, and filter
    tokens = [
        lemmatizer.lemmatize(word)
        for word in tokens
        if len(word) > 2 and
           not any(char.isdigit() for char in word) and
           'ax' not in word and
           not re.search(r'(.)\1{2,}', word) and
           word not in {'wa', 'ha'}
    ]

    # Remove English stopwords
    stop_words = set(stopwords.words('english'))
    stop_words.update([
            'thanks', 'please', 'hi', 'bye', 'regards',
            'hello', 'hey', 'gonna', 'wanna', 'yeah',
            'wa', 'ha', 'doe'])
    tokens = [token for token in tokens if token not in stop_words]

    # Join tokens back into a single string
    cleaned_text = ' '.join(tokens)

    return cleaned_text
# Clean the texts
cleaned_texts = [clean_text(text) for text in train_data.data]

# Convert cleaned_texts to a DataFrame
df_cleaned = pd.DataFrame(cleaned_texts, columns=['cleaned_text'])

# Save to CSV
df_cleaned.to_csv('./assets/cleaned_texts.csv', index=False)
```

3.3 Data Exploration

[5]:
```python
def explore_language():
    """
    Conducts a comprehensive analysis of the 20 Newsgroups dataset, including␣
 ↪data cleansing,
    visualization of document length distribution, topic modeling, and category␣
 ↪distribution.
```

```
    Returns:
        dict: A dictionary containing cleaned texts, topic insights, document␣
↪lengths,
              LDA output, LDA model, feature names, vectorizer, and TF-IDF␣
↪matrix.
    """

    # Display basic metrics
    print("Language Analysis Report")
    print(f"Total Documents: {len(cleaned_texts)}")
    print(f"Number of Categories: {len(train_data.target_names)}")

    # Analyze document lengths
    doc_lengths = [len(text.split()) for text in cleaned_texts]

    # Visualize document length distribution
    fig_length = px.histogram(
        x=doc_lengths,
        nbins=50,
        title='Document Length Distribution',
        labels={'x': 'Number of Words', 'y': 'Number of Documents'},
        color_discrete_sequence=['skyblue'],
        marginal='box'
    )
    fig_length.update_layout(
        title_font_size=18,
        xaxis_title_font_size=14,
        yaxis_title_font_size=14
    )
    fig_length.show()

    # Visualize document length distribution with a range of 0-500 words
    filtered_lengths = [length for length in doc_lengths if length <= 500]
    fig_length_filtered = px.histogram(
        x=filtered_lengths,
        nbins=50,
        title='Document Length Distribution (0-500 Words)',
        labels={'x': 'Number of Words', 'y': 'Number of Documents'},
        color_discrete_sequence=['skyblue'],
        marginal='box',
        range_x=[0, 500]
    )
    fig_length_filtered.update_layout(
        title_font_size=18,
        xaxis_title_font_size=14,
        yaxis_title_font_size=14
```

```python
)
fig_length_filtered.show()

# Vectorize the text using TF-IDF
vectorizer = TfidfVectorizer(
    max_df=0.90,
    min_df=3,
    max_features=10000,
    stop_words='english')
tfidf_matrix = vectorizer.fit_transform(cleaned_texts)

# Perform topic modeling using Latent Dirichlet Allocation (LDA)
lda_model = LatentDirichletAllocation(n_components=6, random_state=42)
lda_output = lda_model.fit_transform(tfidf_matrix)

# Extract feature names for topic analysis
feature_names = vectorizer.get_feature_names_out()

# Visualize category distribution
category_counts = pd.Series(train_data.target).value_counts()
fig_category = px.bar(
    x=category_counts.index,
    y=category_counts.values,
    title='News Group Category Distribution',
    labels={'x': 'Category Index', 'y': 'Number of Documents'},
    color_discrete_sequence=['green']
)
fig_category.update_layout(
    title_font_size=18,
    xaxis_title_font_size=14,
    yaxis_title_font_size=14
)
fig_category.show()

# Get the top keywords for the corpus
tfidf_sums = tfidf_matrix.sum(axis=0).A1
keywords = sorted(zip(tfidf_sums, feature_names), reverse=True)[:10]

# Display the top words for each topic
print("\nTop Words for Each Topic:")
topic_insights = []
for topic_idx, topic in enumerate(lda_model.components_):
    top_features_ind = topic.argsort()[:-11:-1]
    top_features = [feature_names[i] for i in top_features_ind]
    topic_insights.append({
        'theme_number': topic_idx + 1,
        'top_words': top_features
```

```python
        })
        print(f"Topic {topic_idx + 1}: {', '.join(top_features)}")

    # Display comprehensive insights
    print("\nComprehensive Insights")
    print(f"Unique Linguistic Features: {len(feature_names)}")
    print(f"Average Document Length: {np.mean(doc_lengths):.2f} words")
    print(f"Median Document Length: {np.median(doc_lengths):.2f} words")

    return {
        'cleaned_texts': cleaned_texts,
        'topic_insights': topic_insights,
        'document_lengths': doc_lengths,
        'lda_output': lda_output,
        'lda_model': lda_model,
        'feature_names': feature_names,
        'vectorizer': vectorizer,
        'tfidf_matrix': tfidf_matrix,
        'keywords': keywords,
        'fig_length': fig_length,
        'fig_length_filtered': fig_length_filtered,
        'fig_category': fig_category
    }
analysis_results = explore_language()

with open('./assets/analysis_results.pkl', 'wb') as file:
    pickle.dump(analysis_results, file)
```

```
Language Analysis Report
Total Documents: 11314
Number of Categories: 20


Top Words for Each Topic:
Topic 1: people, dont, think, like, god, know, time, say, right, year
Topic 2: drive, card, monitor, scsi, mac, disk, video, controller, port, apple
Topic 3: key, use, know, file, program, like, email, chip, information, space
Topic 4: window, file, server, problem, widget, program, manager, version,
running, use
Topic 5: game, team, player, season, play, hockey, gordon, league, nhl,
surrender
Topic 6: driver, end, card, vlb, diamond, ati, vesa, ultra, stealth, speedstar

Comprehensive Insights
Unique Linguistic Features: 10000
Average Document Length: 91.58 words
Median Document Length: 41.00 words
```

3.4 Interactive Topic Visualization

[6]:
```python
def create_interactive_topic_visualization(analysis_results):
    """
    Generates an interactive t-SNE visualization of topics derived from Latent␣
    ↪Dirichlet Allocation (LDA) output.

    Parameters:
        analysis_results (dict): A dictionary containing analysis results,␣
    ↪including LDA output.

    Returns:
        plotly.graph_objs._figure.Figure: The Plotly figure object representing␣
    ↪the t-SNE visualization.
    """
    # Perform t-SNE dimensionality reduction on the LDA output
    tsne = TSNE(n_components=2, random_state=42)
    tsne_results = tsne.fit_transform(analysis_results['lda_output'])

    # Create a DataFrame for visualization
    df_tsne = pd.DataFrame(tsne_results, columns=['Dimension 1', 'Dimension 2'])
    df_tsne['Topic'] = np.argmax(analysis_results['lda_output'], axis=1) + 1  #␣
    ↪Topics are 1-indexed

    # Generate an interactive scatter plot using Plotly
    fig = px.scatter(
        df_tsne,
        x='Dimension 1',
        y='Dimension 2',
        color='Topic',
        title='Topic Landscape Exploration',
        labels={'Dimension 1': 'Dimension 1', 'Dimension 2': 'Dimension 2'},
        color_continuous_scale=px.colors.sequential.Viridis
    )

    # Update layout for better readability
    fig.update_layout(
        title_font_size=18,
        xaxis_title_font_size=14,
        yaxis_title_font_size=14
    )

    # Display the figure
    fig.show()
    fig.write_html('./assets/topic_landscape.html')
    return fig
```

```
# Execute the visualization
if __name__ == "__main__":
    # Ensure that 'analysis_results' has been previously defined, typically by␣
↪running 'explore_language()'
    try:
        topic_visualization =␣
↪create_interactive_topic_visualization(analysis_results)
    except NameError:
        print("The 'analysis_results' variable is not defined. Please run the␣
↪'explore_language()' function first.")
```

3.5 Interactive Topic Exploration using pyLDAvis

[7]:
```
pyLDAvis.enable_notebook()
panel = pyLDAvis.lda_model.prepare(analysis_results['lda_model'],␣
↪analysis_results['tfidf_matrix'], analysis_results['vectorizer'], mds='tsne')
pyLDAvis.display(panel)
```

[7]: <IPython.core.display.HTML object>

3.6 Evaluations

[8]:
```
def evaluate_lda_model(lda_model, tfidf_matrix, feature_names, n_top_words=5):
    """
    Conducts a comprehensive evaluation of a Latent Dirichlet Allocation (LDA)␣
↪model.

    This function calculates various evaluation metrics such as perplexity,␣
↪topic coherence,
    and topic diversity. It also generates visualizations for topic word␣
↪importance, topic
    similarity, and coherence scores. Additionally, it provides detailed␣
↪summaries of each topic.

    Parameters:
        lda_model (LatentDirichletAllocation): The trained LDA model.
        tfidf_matrix (scipy.sparse.csr_matrix): The TF-IDF feature matrix used␣
↪to train the LDA model.
        feature_names (list of str): The list of feature names corresponding to␣
↪the columns in the TF-IDF matrix.
        n_top_words (int, optional): The number of top words to consider for␣
↪each topic. Defaults to 5.

    Returns:
        dict: A dictionary containing evaluation metrics including perplexity,␣
↪coherence scores,
```

```python
            topic diversity, and detailed topic summaries.
    """

    # 1. Calculate Perplexity Score
    perplexity = lda_model.perplexity(tfidf_matrix)

    # 2. Calculate Topic Coherence
    def calculate_topic_coherence(model, feature_names, n_top_words=10):
        """
        Calculates the coherence scores for each topic in the LDA model.

        Parameters:
            model (LatentDirichletAllocation): The trained LDA model.
            feature_names (list of str): The list of feature names.
            n_top_words (int): The number of top words to consider for coherence calculation.

        Returns:
            list of float: Coherence scores for each topic.
        """
        coherence_scores = []
        for topic_idx, topic in enumerate(model.components_):
            top_words_indices = topic.argsort()[:-n_top_words - 1:-1]
            top_words = [feature_names[i] for i in top_words_indices]

            word_coherence = []
            for i in range(len(top_words)):
                for j in range(i + 1, len(top_words)):
                    # Calculate coherence based on shared characters (simple example)
                    word_coherence.append(1 if len(set(top_words[i]) & set(top_words[j])) > 0 else 0)

            coherence_scores.append(np.mean(word_coherence) if word_coherence else 0)

        return coherence_scores

    # 3. Calculate Topic Diversity
    def calculate_topic_diversity(model, feature_names, n_top_words=10):
        """
        Calculates the diversity of topics based on the uniqueness of their top words.

        Parameters:
            model (LatentDirichletAllocation): The trained LDA model.
            feature_names (list of str): The list of feature names.
```

```
        n_top_words (int): The number of top words to consider for␣
↪diversity calculation.

    Returns:
        dict: A dictionary containing the number of unique words, total␣
↪words, and diversity ratio.
    """
    all_top_words = []
    for topic in model.components_:
        top_words_indices = topic.argsort()[:-n_top_words - 1:-1]
        top_words = [feature_names[i] for i in top_words_indices]
        all_top_words.extend(top_words)

    unique_words = len(set(all_top_words))
    total_words = len(all_top_words)

    return {
        'unique_words': unique_words,
        'total_words': total_words,
        'diversity_ratio': unique_words / total_words if total_words > 0␣
↪else 0
    }

# 4. Prepare Topic Summaries for Interpretability
def prepare_topic_summaries(model, feature_names, n_top_words=10):
    """
    Prepares summaries of each topic, including top words and their␣
↪corresponding weights.

    Parameters:
        model (LatentDirichletAllocation): The trained LDA model.
        feature_names (list of str): The list of feature names.
        n_top_words (int): The number of top words to include in each␣
↪summary.

    Returns:
        tuple: A tuple containing a list of topic summaries and data for␣
↪visualization.
    """
    topic_summaries = []
    topic_data_for_visualization = []

    for topic_idx, topic in enumerate(model.components_):
        top_words_indices = topic.argsort()[:-n_top_words - 1:-1]
        top_words = [feature_names[i] for i in top_words_indices]
        top_weights = topic[top_words_indices]
```

```
        # Prepare data for visualization
        topic_data_for_visualization.extend([
            {
                'Topic': f'Topic {topic_idx + 1}',
                'Word': word,
                'Weight': weight
            }
            for word, weight in zip(top_words, top_weights)
        ])

        # Create topic summary
        topic_summaries.append({
            'topic_number': topic_idx + 1,
            'top_words': top_words,
            'top_weights': top_weights.tolist()
        })

    return topic_summaries, topic_data_for_visualization

# 5. Calculate Topic Similarity
def calculate_topic_similarity(model):
    """
    Calculates the cosine similarity between topics based on their word␣
↪distributions.

    Parameters:
        model (LatentDirichletAllocation): The trained LDA model.

    Returns:
        numpy.ndarray: A matrix representing the similarity between each␣
↪pair of topics.
    """
    topic_similarities = cosine_similarity(model.components_)
    return topic_similarities

# Perform Evaluations
coherence_scores = calculate_topic_coherence(lda_model, feature_names,␣
↪n_top_words)
topic_diversity = calculate_topic_diversity(lda_model, feature_names,␣
↪n_top_words)
topic_summaries, topic_data_for_visualization =␣
↪prepare_topic_summaries(lda_model, feature_names, n_top_words)
topic_similarities = calculate_topic_similarity(lda_model)

# Visualizations
```

```python
# 1. Topic Word Importance
fig_topic_words = px.bar(
    pd.DataFrame(topic_data_for_visualization),
    x='Word',
    y='Weight',
    color='Topic',
    title='Topic Word Importance',
    labels={'Weight': 'Word Weight'},
    height=600,
    width=1200
)
fig_topic_words.update_layout(
    xaxis_tickangle=-45,
    title_font_size=18,
    xaxis_title_font_size=14,
    yaxis_title_font_size=14
)
fig_topic_words.show()

# 2. Topic Similarity Heatmap
fig_topic_similarity = px.imshow(
    topic_similarities,
    title='Topic Similarity Heatmap',
    labels=dict(x="Topics", y="Topics", color="Similarity"),
    color_continuous_scale='Viridis',
    height=800,
    width=800
)
fig_topic_similarity.update_layout(
    title_font_size=18,
    xaxis_title_font_size=14,
    yaxis_title_font_size=14
)
fig_topic_similarity.show()

# 3. Coherence Scores Visualization
fig_coherence = px.bar(
    x=[f'Topic {i+1}' for i in range(len(coherence_scores))],
    y=coherence_scores,
    title='Topic Coherence Scores',
    labels={'x': 'Topics', 'y': 'Coherence Score'},
    height=500,
    width=800
)
fig_coherence.update_layout(
    title_font_size=18,
    xaxis_title_font_size=14,
```

```python
        yaxis_title_font_size=14
    )
    fig_coherence.show()

    # Print Evaluation Summary
    print("\nLDA Model Evaluation Report")
    print(f"Perplexity Score: {perplexity:.2f}")
    print(f"Average Topic Coherence: {np.mean(coherence_scores):.4f}")
    print("Topic Diversity:")
    print(f"  - Unique Words: {topic_diversity['unique_words']}")
    print(f"  - Total Words: {topic_diversity['total_words']}")
    print(f"  - Diversity Ratio: {topic_diversity['diversity_ratio']:.4f}")

    # Detailed Topic Summaries
    print("\nTopic Summaries:")
    for summary in topic_summaries:
        print(f"\nTopic {summary['topic_number']}:")
        for word, weight in zip(summary['top_words'], summary['top_weights']):
            print(f"  {word} (Weight: {weight:.4f})")

    return {
        'perplexity': perplexity,
        'coherence_scores': coherence_scores,
        'topic_diversity': topic_diversity,
        'topic_summaries': topic_summaries
    }

# Execute the Evaluation
if __name__ == "__main__":
    # Ensure that 'analysis_results' has been previously defined, typically by␣
 ↪running 'explore_language_kingdom()'
    try:
        lda_evaluation = evaluate_lda_model(
            analysis_results['lda_model'],
            analysis_results['tfidf_matrix'],
            analysis_results['feature_names'],
            n_top_words=10
        )
    except NameError:
        print("The 'analysis_results' variable is not defined. Please run the␣
 ↪'explore_language()' function first.")
```

```
LDA Model Evaluation Report
Perplexity Score: 12369.29
Average Topic Coherence: 0.7815
Topic Diversity:
```

```
  - Unique Words: 54
  - Total Words: 60
  - Diversity Ratio: 0.9000

Topic Summaries:

Topic 1:
  people (Weight: 125.0549)
  dont (Weight: 115.5123)
  think (Weight: 107.3700)
  like (Weight: 99.8436)
  god (Weight: 95.7733)
  know (Weight: 92.4745)
  time (Weight: 90.9770)
  say (Weight: 87.1747)
  right (Weight: 81.0975)
  year (Weight: 78.9608)

Topic 2:
  drive (Weight: 71.9383)
  card (Weight: 62.4598)
  monitor (Weight: 42.3347)
  scsi (Weight: 36.9458)
  mac (Weight: 32.4266)
  disk (Weight: 31.0127)
  video (Weight: 29.1977)
  controller (Weight: 28.1581)
  port (Weight: 27.3119)
  apple (Weight: 26.4336)

Topic 3:
  key (Weight: 64.7250)
  use (Weight: 56.8545)
  know (Weight: 56.6855)
  file (Weight: 54.9257)
  program (Weight: 53.6102)
  like (Weight: 52.3435)
  email (Weight: 49.7191)
  chip (Weight: 44.2721)
  information (Weight: 42.4681)
  space (Weight: 40.5469)

Topic 4:
  window (Weight: 60.2073)
  file (Weight: 39.4453)
  server (Weight: 23.0742)
  problem (Weight: 20.5945)
  widget (Weight: 17.3565)
```

```
  program (Weight: 16.5639)
  manager (Weight: 13.9742)
  version (Weight: 13.9636)
  running (Weight: 13.2786)
  use (Weight: 13.0565)

Topic 5:
  game (Weight: 41.2786)
  team (Weight: 40.5616)
  player (Weight: 30.7768)
  season (Weight: 20.4246)
  play (Weight: 18.7796)
  hockey (Weight: 17.8131)
  gordon (Weight: 17.0507)
  league (Weight: 16.4396)
  nhl (Weight: 15.8548)
  surrender (Weight: 15.5767)

Topic 6:
  driver (Weight: 17.7445)
  end (Weight: 14.1934)
  card (Weight: 10.3316)
  vlb (Weight: 10.1027)
  diamond (Weight: 9.2351)
  ati (Weight: 8.9447)
  vesa (Weight: 5.9634)
  ultra (Weight: 5.8850)
  stealth (Weight: 5.4574)
  speedstar (Weight: 5.2910)
```

4 Level 2

4.1 Keyword Extraction using TF-IDF

[9]:
```
analysis_results['keywords']
```

[9]:
```
[(184.72498884111226, 'know'),
 (178.87880861504232, 'like'),
 (170.86841045827614, 'dont'),
 (149.35114533507405, 'people'),
 (149.20791794601638, 'think'),
 (135.10035779107653, 'time'),
 (130.89169284178772, 'use'),
 (125.4596023810102, 'good'),
 (120.86957670016423, 'problem'),
 (118.54215403181237, 'window')]
```

4.2 Sentiment Analysis

[10]:
```
def analyze_sentiments(texts):
    """
    Perform sentiment analysis on a list of texts.
    Returns a list of sentiment scores.
    """
    sentiments = []
    for text in texts:
        blob = TextBlob(text)
        sentiments.append(blob.sentiment.polarity)  # Polarity score (-1 to 1)
    return sentiments

# Analyze sentiments for the cleaned texts
sentiments = analyze_sentiments(cleaned_texts)
print(f"Average Sentiment Polarity: {sum(sentiments)/len(sentiments):.2f}")
```

```
Average Sentiment Polarity: 0.08
```

4.3 Generate a Word Cloud

[11]:
```
def generate_wordcloud(texts):
    """
    Generate and display a word cloud from the text corpus.
    """
    # Combine texts
    combined_text = ' '.join(texts)

    # Create a word cloud image
    wordcloud = WordCloud(width=800, height=400, background_color='white').
 ↪generate(combined_text)

    # Convert the WordCloud to an RGB array
    wordcloud_image = wordcloud.to_array()  # Produces an (H, W, 3) numpy array

    # Use Plotly's imshow to display the image
    fig = px.imshow(wordcloud_image)
    fig.update_layout(
        title="Word Cloud",
        title_font_size=20,
        xaxis=dict(showgrid=False, showticklabels=False, zeroline=False),
        yaxis=dict(showgrid=False, showticklabels=False, zeroline=False),
        coloraxis_showscale=False
    )
    fig.write_html("./assets/wordcloud.html")
    fig.show()

# Generate the word cloud
```

```
generate_wordcloud(cleaned_texts)
```

4.4 Semantic Analysis with Named Entity Recognition

[12]:
```
def named_entity_recognition(text):
    """
    Perform named entity recognition using NLTK.
    """
    sentences = sent_tokenize(text)
    for sent in sentences:
        tokens = word_tokenize(sent)
        tagged = nltk.pos_tag(tokens)
        entities = nltk.ne_chunk(tagged)
        print(entities)

# Example: Run NER on the first document
named_entity_recognition(cleaned_texts[0])
```

```
(S
  wondering/VBG
  anyone/NN
  could/MD
  enlighten/VB
  car/NN
  saw/JJ
  day/NN
  door/VB
  sport/NN
  car/NN
  looked/VBD
  late/RB
  early/RB
  called/VBN
  bricklin/NN
  door/NN
  really/RB
  small/JJ
  addition/NN
  front/JJ
  bumper/NN
  separate/JJ
  rest/NN
  body/NN
  know/VBP
  anyone/NN
  tellme/JJ
  model/NN
```

```
name/NN
engine/NN
spec/JJ
year/NN
production/NN
car/NN
made/VBD
history/NN
whatever/WDT
info/NN
funky/NN
looking/VBG
car/NN
email/NN)
```

4.4.1 Spacy Named Entity Recognition

[13]:
```python
def install_spacy_model():
    """
    Install spaCy English language model if not already installed.
    """
    import subprocess
    import sys
    try:
        import spacy
        spacy.load('en_core_web_sm')
    except OSError:
        print("Downloading spaCy English model...")
        subprocess.check_call([sys.executable, "-m", "spacy", "download",␣
↪"en_core_web_sm"])

def spacy_named_entity_recognition(text):
    """
    Perform Named Entity Recognition using spaCy.

    Args:
        text (str): Input text to analyze

    Returns:
        list: Named entities found in the text
    """
    # Ensure spaCy model is installed
    install_spacy_model()

    # Load the English language model
    nlp = spacy.load('en_core_web_sm')
```

```python
    # Process the text
    doc = nlp(text)

    # Extract named entities
    entities = []
    for ent in doc.ents:
        entities.append({
            'text': ent.text,
            'label': ent.label_,
            'start_char': ent.start_char,
            'end_char': ent.end_char
        })

    return entities

def visualize_spacy_entities(entities):
    """
    Visualize spaCy named entities with detailed information.

    Args:
        entities (list): List of named entities
    """
    # Group entities by type
    entities_by_type = {}
    for entity in entities:
        entity_type = entity['label']
        if entity_type not in entities_by_type:
            entities_by_type[entity_type] = []
        entities_by_type[entity_type].append(entity['text'])

    print("  Named Entities Discovered:")

    # Print entities grouped by type
    for etype, elist in entities_by_type.items():
        print(f"• {etype}:")
        for e in set(elist):
            print(f"  - {e}")

    # Print summary statistics
    print("\n  Entity Type Summary:")
    for etype, elist in entities_by_type.items():
        print(f"- {etype}: {len(set(elist))} unique entities")

def demonstrate_spacy_ner(text):
    """
    Comprehensive Named Entity Recognition demonstration using spaCy.
```

```
    Args:
        text (str): Input text to analyze

    Returns:
        list: Extracted named entities
    """
    # Perform Named Entity Recognition
    entities = spacy_named_entity_recognition(text)

    # Visualize entities
    visualize_spacy_entities(entities)

    return entities

# SpaCy Entity Types Explanation
SPACY_ENTITY_TYPES = {
    'PERSON': 'People, including fictional',
    'NORP': 'Nationalities or religious/political groups',
    'FAC': 'Buildings, airports, highways, bridges, etc.',
    'ORG': 'Companies, agencies, institutions',
    'GPE': 'Countries, cities, states',
    'LOC': 'Non-GPE locations, mountain ranges, bodies of water',
    'PRODUCT': 'Objects, vehicles, foods, etc. (Not services)',
    'EVENT': 'Named hurricanes, battles, wars, sports events',
    'WORK_OF_ART': 'Titles of books, songs, etc.',
    'LAW': 'Named documents made into laws',
    'LANGUAGE': 'Any named language',
    'DATE': 'Absolute or relative dates or periods',
    'TIME': 'Times smaller than a day',
    'PERCENT': 'Percentage, including "%"',
    'MONEY': 'Monetary values, including unit',
    'QUANTITY': 'Measurements, as of weight or distance',
    'ORDINAL': '"first", "second", etc.',
    'CARDINAL': 'Numerals that do not fall into another type'
}

# Print Entity Type Explanations
print("\n SpaCy Named Entity Types:")
for code, description in SPACY_ENTITY_TYPES.items():
    print(f"{code}: {description}")

# Attempt to use the original preprocessed text
try:
    sample_text = cleaned_texts[0]
    print("\n Analyzing First Document:")
    entities = demonstrate_spacy_ner(sample_text)
except Exception as e:
```

```
    print(f"Error with preprocessed text: {e}")

    # Fallback sample text
    sample_text = cleaned_texts[0]

    print("\n  Using Fallback Sample Text:")
    entities = demonstrate_spacy_ner(sample_text)
```

```
  SpaCy Named Entity Types:
PERSON: People, including fictional
NORP: Nationalities or religious/political groups
FAC: Buildings, airports, highways, bridges, etc.
ORG: Companies, agencies, institutions
GPE: Countries, cities, states
LOC: Non-GPE locations, mountain ranges, bodies of water
PRODUCT: Objects, vehicles, foods, etc. (Not services)
EVENT: Named hurricanes, battles, wars, sports events
WORK_OF_ART: Titles of books, songs, etc.
LAW: Named documents made into laws
LANGUAGE: Any named language
DATE: Absolute or relative dates or periods
TIME: Times smaller than a day
PERCENT: Percentage, including "%"
MONEY: Monetary values, including unit
QUANTITY: Measurements, as of weight or distance
ORDINAL: "first", "second", etc.
CARDINAL: Numerals that do not fall into another type

  Analyzing First Document:
  Named Entities Discovered:
• GPE:
  - bricklin

  Entity Type Summary:
- GPE: 1 unique entities
```

5 Level 3

5.1 Text Generator

[18]:
```
class TextGenerator:
    def __init__(self, texts, max_features=2000, model_save_dir='./
 ↪model_artifacts'):
        """
        Initialize the text generation model with preprocessed texts
```

```
        Args:
            texts (list): List of preprocessed text documents
            max_features (int): Maximum number of features to consider
            model_save_dir (str): Directory to save model and tokenizer
        """
        # Ensure texts is a list and not empty
        self.texts = texts if texts else ["default text for generation"]
        self.max_features = max_features
        self.device = torch.device('cuda' if torch.cuda.is_available() else␣
↪'cpu')

        # Create directory for model artifacts if it doesn't exist
        self.model_save_dir = model_save_dir
        os.makedirs(self.model_save_dir, exist_ok=True)

        # Initialize attributes
        self.tokenizer = None
        self.model = None
        self.tfidf_vectorizer = analysis_results['vectorizer']
        self.tfidf_matrix = analysis_results['tfidf_matrix']
        self.feature_names = analysis_results['feature_names']
        self.word_probabilities = None

    def save_model(self, model_name='fine_tuned_model'):
        """
        Enhanced model saving with special token handling
        """
        if not self.model or not self.tokenizer:
            raise ValueError("Model not fine-tuned. Call fine_tune_gpt2() first.
↪")

        # Create full path for saving
        save_path = os.path.join(self.model_save_dir, model_name)
        os.makedirs(save_path, exist_ok=True)

        try:
            # Save model and tokenizer
            self.model.save_pretrained(save_path)
            self.tokenizer.save_pretrained(save_path)

            print(f"Model and tokenizer saved successfully to {save_path}")

            # Verify saved tokenizer configuration
            print("Saved Pad Token:", self.tokenizer.pad_token)
            print("Saved Pad Token ID:", self.tokenizer.pad_token_id)

            return save_path
```

```
        except Exception as e:
            print(f"Error saving model: {e}")
            raise

    def load_model(self, model_name='fine_tuned_model'):
        """
        Enhanced model loading with robust tokenizer setup
        """
        # Construct full path to the saved model
        load_path = os.path.join(self.model_save_dir, model_name)

        try:
            # Load tokenizer and model
            self.tokenizer = GPT2Tokenizer.from_pretrained(load_path)
            self.model = GPT2LMHeadModel.from_pretrained(load_path)

            # Ensure pad token is set uniquely
            if self.tokenizer.pad_token is None:
                self.tokenizer.add_special_tokens({
                    'pad_token': '[PAD]'
                })
                self.model.resize_token_embeddings(len(self.tokenizer))

            # Move model to device
            self.model = self.model.to(self.device)

            print(f"Model loaded successfully from {load_path}")

            # Verify tokenizer configuration
            print("Pad Token:", self.tokenizer.pad_token)
            print("Pad Token ID:", self.tokenizer.pad_token_id)
            print("EOS Token:", self.tokenizer.eos_token)
            print("EOS Token ID:", self.tokenizer.eos_token_id)

            return self.model, self.tokenizer
        except Exception as e:
            print(f"Error loading model: {e}")
            raise

    def fine_tune_gpt2(self,
                       model_name='distilgpt2',
                       epochs=1,
                       batch_size=4,
                       save_after_training=True):
        """
        Improved GPT-2 fine-tuning with robust tokenizer handling
        """
```

```
        # Initialize tokenizer and model
        self.tokenizer = GPT2Tokenizer.from_pretrained(model_name)

        # Create a unique pad token
        self.tokenizer.add_special_tokens({
            'pad_token': '[PAD]'
        })

        # Resize model embeddings to match new tokenizer
        self.model = GPT2LMHeadModel.from_pretrained(model_name)
        self.model.resize_token_embeddings(len(self.tokenizer))

        # Move model to device
        self.model = self.model.to(self.device)

        # Prepare texts for training
        truncated_texts = [text[:500] for text in self.texts]

        # Tokenize texts with explicit padding and attention mask
        encodings = self.tokenizer(
            truncated_texts,
            truncation=True,
            max_length=128,
            padding=True,  # Ensure padding
            return_tensors='pt'
        )

        # Prepare dataset
        input_ids = encodings['input_ids'][:500].to(self.device)
        attention_mask = encodings['attention_mask'][:500].to(self.device)

        # Create data loader
        dataset = torch.utils.data.TensorDataset(input_ids, attention_mask)
        dataloader = torch.utils.data.DataLoader(dataset,␣
↪batch_size=batch_size, shuffle=True)

        # Training setup
        optimizer = torch.optim.AdamW(self.model.parameters(), lr=5e-5)
        self.model.train()

        # Training loop
        for epoch in range(epochs):
            total_loss = 0
            for batch_idx, batch in enumerate(dataloader):
                inputs, masks = batch

                optimizer.zero_grad()
```

```
                outputs = self.model(
                    input_ids=inputs,
                    attention_mask=masks,
                    labels=inputs
                )
                loss = outputs.loss
                loss.backward()
                optimizer.step()

                total_loss += loss.item()

                # Progress tracking
                if (batch_idx + 1) % 10 == 0:
                    print(f"Epoch {epoch+1}, Batch {batch_idx+1}/
↪{len(dataloader)}, Loss: {loss.item():.4f}")

            print(f"Epoch {epoch+1} Average Loss: {total_loss/len(dataloader):.
↪4f}")

        # Optional saving after training
        if save_after_training:
            self.save_model()

        return self.model

    def generate_text(self, prompt="The", max_length=50):
        """
        Generate text using the fine-tuned model
        """
        if not self.model or not self.tokenizer:
            # Attempt to load the last saved model
            try:
                self.load_model()
            except Exception as e:
                raise ValueError("No model available. Fine-tune a model first.
↪") from e

        # Prepare for generation
        self.model.eval()
        self.model.to(self.device)

        # Ensure prompt is not empty
        prompt = prompt or "The"

        # Generate text
        input_ids = self.tokenizer.encode(prompt, return_tensors='pt').to(self.
↪device)
```

```
        with torch.no_grad():
            output = self.model.generate(
                input_ids,
                max_length=max_length,
                num_return_sequences=1,
                no_repeat_ngram_size=2,
                do_sample=True,
                top_k=50,
                top_p=0.95,
                pad_token_id=self.tokenizer.eos_token_id
            )

        return self.tokenizer.decode(output[0], skip_special_tokens=True)

def main(sample_texts=None):
    # Default sample texts if none provided
    if not sample_texts or not isinstance(sample_texts, list):
        sample_texts = [
            "Information technology is rapidly evolving.",
            "Machine learning continues to advance.",
            "Artificial intelligence has many applications."
        ]

    # Create generator
    generator = TextGenerator(sample_texts)

    try:
        # Fine-tune and save model
        generator.fine_tune_gpt2(epochs=3)

        # Generate text from the trained model
        generated_text = generator.generate_text(
            prompt="Information technology",
            max_length=50
        )
        print("\nGenerated Text:")
        print(generated_text)

        # Demonstrate model loading
        print("\nTesting Model Loading...")
        new_generator = TextGenerator(sample_texts)
        loaded_model, loaded_tokenizer = new_generator.load_model()

        # Generate text from loaded model
        loaded_text = new_generator.generate_text(
            prompt="Artificial intelligence",
```

```python
            max_length=50
        )
        print("\nText from Loaded Model:")
        print(loaded_text)

    except Exception as e:
        print(f"An error occurred: {e}")
        import traceback
        traceback.print_exc()

# Run main with existing texts or default texts
try:
    main(cleaned_texts)
except NameError:
    main()
```

```
Epoch 1, Batch 10/125, Loss: 5.9523
Epoch 1, Batch 20/125, Loss: 6.0803
Epoch 1, Batch 30/125, Loss: 4.7639
Epoch 1, Batch 40/125, Loss: 4.1307
Epoch 1, Batch 50/125, Loss: 4.6495
Epoch 1, Batch 60/125, Loss: 3.1746
Epoch 1, Batch 70/125, Loss: 2.8280
Epoch 1, Batch 80/125, Loss: 4.3462
Epoch 1, Batch 90/125, Loss: 2.9588
Epoch 1, Batch 100/125, Loss: 2.3487
Epoch 1, Batch 110/125, Loss: 4.5605
Epoch 1, Batch 120/125, Loss: 2.0487
Epoch 1 Average Loss: 4.4384
Epoch 2, Batch 10/125, Loss: 3.6887
Epoch 2, Batch 20/125, Loss: 2.9031
Epoch 2, Batch 30/125, Loss: 3.0917
Epoch 2, Batch 40/125, Loss: 3.2942
Epoch 2, Batch 50/125, Loss: 1.6688
Epoch 2, Batch 60/125, Loss: 3.0164
Epoch 2, Batch 70/125, Loss: 3.3250
Epoch 2, Batch 80/125, Loss: 2.9569
Epoch 2, Batch 90/125, Loss: 2.1095
Epoch 2, Batch 100/125, Loss: 1.9904
Epoch 2, Batch 110/125, Loss: 2.5639
Epoch 2, Batch 120/125, Loss: 3.5705
Epoch 2 Average Loss: 2.9753
Epoch 3, Batch 10/125, Loss: 1.9355
Epoch 3, Batch 20/125, Loss: 1.9614
Epoch 3, Batch 30/125, Loss: 1.5064
Epoch 3, Batch 40/125, Loss: 2.3451
Epoch 3, Batch 50/125, Loss: 2.1188
```

```
Epoch 3, Batch 60/125, Loss: 3.7380
Epoch 3, Batch 70/125, Loss: 2.4172
Epoch 3, Batch 80/125, Loss: 1.8705
Epoch 3, Batch 90/125, Loss: 3.5383
Epoch 3, Batch 100/125, Loss: 3.6457
Epoch 3, Batch 110/125, Loss: 3.2149
Epoch 3, Batch 120/125, Loss: 2.7539
Epoch 3 Average Loss: 2.7990
Model and tokenizer saved successfully to ./model_artifacts/fine_tuned_model
Saved Pad Token: [PAD]
Saved Pad Token ID: 50257

Generated Text:
Information technology like system know device technology that is able get input
input device type contact information hardware also connect device phone device
controller hardware hardware controller platform controller device platform
platform hardware connect phone phone controller software controller controller
data controller computer monitor controller modem hardware connected

Testing Model Loading…
Model loaded successfully from ./model_artifacts/fine_tuned_model
Pad Token: [PAD]
Pad Token ID: 50257
EOS Token: <|endoftext|>
EOS Token ID: 50256

Text from Loaded Model:
Artificial intelligence is complex knowledge algorithm is known machine code
code algorithm problem code language language software software program software
code software analysis problem software evaluation software problem machine
software design problem development software development system software testing
software application software software understanding system program program
```