```python
import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow.keras.layers import Input, Embedding, Dense, MultiHeadAttention, GlobalMaxPooling1D, LayerNormalization, Dropout
from tensorflow.keras.initializers import Constant
from tensorflow.keras.regularizers import l2
from tensorflow.keras import Model
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from keras.preprocessing.sequence import pad_sequences
from keras.callbacks import EarlyStopping
from keras.optimizers import Adam
from gensim.models import Word2Vec

# Load dataset
train_df = pd.read_csv("tinder.csv")

# Text Preprocessing Functions
import re
import string
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer
from bs4 import BeautifulSoup

nltk.download('stopwords')
nltk.download('punkt')
nltk.download('wordnet')

lemmatizer = WordNetLemmatizer()

def preprocess_text(text):
    text = text.lower()
    tokens = word_tokenize(text)
    stopwords_set = set(stopwords.words('english'))
    tokens = [lemmatizer.lemmatize(word) for word in tokens if word not in stopwords_set]
    return ' '.join(tokens)

def preprocess_text2(text):
    text = text.lower()
    tokens = word_tokenize(text)
    return tokens

def clean_text(text):
    text = BeautifulSoup(text, "html.parser").get_text()
    text = re.sub(r'http\S+', '', text)
    text = re.sub(r'[^\x00-\x7F.]', ' ', text)
    text = re.sub(f'[{re.escape(string.punctuation.replace(".", ""))}]', '', text)
    text = re.sub(r'\b\d+\b', '', text)
    text = re.sub(r'\.{2,}', ' ', text)
    text = re.sub(r'(?<=\.)\s+', ' ', text).strip()
    return text

def remove_repeated_text(text):
    pattern = r'\b(\w+\s?)(\.\s?\1){2,}\b'
    return re.sub(pattern, '', text)

def remove_repeated_text2(text):
    pattern = re.compile(r'\b(\w+)\b\s+\1(?:\s+\1)+\b', re.IGNORECASE)
    def remove_repeats(match):
        return match.group(1)
    return pattern.sub(remove_repeats, text)

def remove_repeating_pattern(text):
    pattern = r'(\w)\1+'
    return re.sub(pattern, '', text)

```

```
# Ensure all values in 'content' are strings and handle missing values
train_df["content"] = train_df["content"].fillna("").astype(str)

# Apply preprocessing
train_df["remove_repeat_word"] = train_df["content"].apply(remove_repeated_text).apply(remove_repeated_text2)
train_df["clean_text"] = train_df["remove_repeat_word"].apply(clean_text).apply(remove_repeating_pattern)
train_df["text_prepro"] = train_df["clean_text"].apply(preprocess_text)


import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Vocabulary calculation functions
def calculate_vocabulary(tokens, N):
    """Calculate vocabulary size for the first N tokens."""
    return len(set(tokens[:N]))

# Vocabulary growth models
def heaps_law(N, k=20, beta=0.6):
    """Vocabulary size based on Heap's law (LNRE)."""
    return k * (N ** beta)

def lstm_vocab_growth(N):
    """
    Simulate LSTM vocabulary growth.
    This assumes diminishing returns due to training bias toward frequent words.
    """
    return np.log(N) ** 2

def bert_vocab_growth(N, max_vocab=30000):
    """
    Simulate BERT vocabulary growth.
    BERT relies on subword tokenization, so the vocabulary growth saturates early.
    """
    return max_vocab * (1 - np.exp(-N / max_vocab))

def laplace_vocab_growth(N, alpha=1, total_vocab=5000):
    """
    Simulate vocabulary growth using Laplace smoothing.
    """
    return total_vocab * (1 - np.exp(-N / (alpha * total_vocab)))

def katz_vocab_growth(N, d=0.5, total_vocab=5000):
    """
    Simulate vocabulary growth using Katz Backoff.
    """
    return total_vocab * (1 - d * np.exp(-N / (total_vocab)))

train_df = pd.DataFrame({"text_prepro": text_prepro_list})

# Tokenize the text data
tokens = []
train_df["text_prepro"].dropna().apply(lambda x: tokens.extend(x.split()))

# Token counts for analysis
N_values = np.logspace(3, 6, num=50, dtype=int)  # Token counts from 1,000 to 1,000,000

# Calculate actual vocabulary sizes
actual_vocab_sizes = [calculate_vocabulary(tokens, N) for N in N_values]

# Simulate LNRE (Heap's Law), LSTM, and BERT growth for comparison
lnre_vocab = [heaps_law(N) for N in N_values]
lstm_vocab = [lstm_vocab_growth(N) for N in N_values]
bert_vocab = [bert_vocab_growth(N) for N in N_values]

# Plotting the results
```

```python
plt.figure(figsize=(12, 6))
plt.plot(N_values, lnre_vocab, label="LNRE (Heap's Law)", color="black", linestyle="solid", linewidth=2)
plt.plot(N_values, lstm_vocab, label="LSTM", color="black", linestyle="dotted", linewidth=2)
plt.plot(N_values, bert_vocab, label="BERT", color="black", linestyle="dashed", linewidth=2)
plt.plot(N_values, actual_vocab_sizes, label="Dataset (Actual)", linestyle="dashdot", color="black", linewidth=2)

# Log scale for better visualization
plt.xscale("log")
plt.yscale("log")

# Adding labels and legend
plt.xlabel("Number of Tokens (N)", fontsize=12)
plt.ylabel("Vocabulary Size (V)", fontsize=12)
plt.title("Vocabulary Growth Comparison: LNRE vs LSTM vs BERT", fontsize=14)
plt.legend(fontsize=12)
plt.grid(True, which="both", linestyle="--", linewidth=0.5)
plt.tight_layout()
plt.savefig("vocabulary_growth_comparison_test_1.png", dpi=300)
plt.show()


import numpy as np
import matplotlib.pyplot as plt
from collections import Counter

# Example: Simulate tokenized data (replace with your actual tokenized dataset)
tokens = ["word1", "word2", "word3", "word1", "word2", "word4", "word1", "word5"]
token_counts = Counter(tokens)

# Sort tokens by frequency
sorted_token_counts = sorted(token_counts.values(), reverse=True)

# Plot the frequency distribution
plt.figure(figsize=(10, 6))
plt.plot(sorted_token_counts, label="Token Frequency Distribution", color = "black")
plt.xlabel("Token Rank")
plt.ylabel("Frequency")
plt.title("Token Frequency Distribution")
plt.grid(True)
plt.tight_layout()
plt.savefig("tail_behav.png", dpi=300)
plt.show()

# Calculate token ranks
ranks = np.arange(1, len(sorted_token_counts) + 1)

# Plot on a log-log scale
plt.figure(figsize=(10, 6))
plt.loglog(ranks, sorted_token_counts, marker="o", color = "black", label="Rank-Frequency Plot")
plt.xlabel("Rank (Log Scale)")
plt.ylabel("Frequency (Log Scale)")
plt.title("Log-Log Plot of Token Rank vs. Frequency")
plt.grid(True, which="both", linestyle="--", linewidth=0.5)
plt.tight_layout()
plt.savefig("tail_behav_2.png", dpi=300)
plt.show()
plt.legend()
plt.show()

import numpy as np
import matplotlib.pyplot as plt
from collections import Counter

# Define vocabulary growth models
```

```python
def heaps_law(N, k=20, beta=0.6):
    return k * (N ** beta)

def lstm_vocab_growth(N):
    return np.log(N) ** 2

def bert_vocab_growth(N, max_vocab=30000):
    return max_vocab * (1 - np.exp(-N / max_vocab))

def laplace_vocab_growth(N, alpha=1, total_vocab=5000):
    return total_vocab * (1 - np.exp(-N / (alpha * total_vocab)))

def katz_vocab_growth(N, d=0.5, total_vocab=5000):
    return total_vocab * (1 - d * np.exp(-N / (total_vocab)))

def calculate_vocabulary(tokens, N):
    return len(set(tokens[:N]))

# Simulated token counts
N_values = np.logspace(3, 6, num=50, dtype=int)

# Ensure tokens are defined (replace with your actual data extraction)
tokens = []
train_df["text_prepro"].dropna().apply(lambda x: tokens.extend(x.split()))

# Plotting all vocabulary growth models
plt.figure(figsize=(12, 6))

# LNRE (Heap's Law)
lnre_vocab = [heaps_law(N) for N in N_values]
plt.plot(N_values, lnre_vocab, label="LNRE (Heap's Law)", color="black", linestyle = "solid", linewidth=2)

# LSTM Vocabulary Growth
lstm_vocab = [lstm_vocab_growth(N) for N in N_values]
plt.plot(N_values, lstm_vocab, label="LSTM", color="black", linestyle="dashed", linewidth=2)

# BERT Vocabulary Growth
bert_vocab = [bert_vocab_growth(N) for N in N_values]
plt.plot(N_values, bert_vocab, label="BERT", color="black", linestyle= "-", linewidth=2)

# Laplace Smoothing Vocabulary Growth
laplace_vocab = [laplace_vocab_growth(N) for N in N_values]
plt.plot(N_values, laplace_vocab, label="Laplace Smoothing", color="black", linestyle=":" ,  linewidth=2)

# Katz Backoff Vocabulary Growth
katz_vocab = [katz_vocab_growth(N) for N in N_values]
plt.plot(N_values, katz_vocab, label="Katz Backoff", color="black", linestyle="-.", linewidth=2)

# Adding actual vocabulary sizes
actual_vocab_sizes = [calculate_vocabulary(tokens, N) for N in N_values]
plt.plot(N_values, actual_vocab_sizes, label="Dataset (Actual)", color="black", linestyle="--", linewidth=2)

# Log scale for better visualization
plt.xscale("log")
plt.yscale("log")

# Adding labels and legend
plt.xlabel("Number of Tokens (N)", fontsize=12)
plt.ylabel("Vocabulary Size (V)", fontsize=12)
plt.title("Vocabulary Growth Comparison: All Models", fontsize=14)
plt.legend(fontsize=12)
plt.grid(True, which="both", linestyle="--", linewidth=0.5)
plt.tight_layout()
plt.savefig("vocabulary_growth_comparison_all_models.png", dpi=300)
```

```
plt.show()

import numpy as np
import matplotlib.pyplot as plt

# Define vocabulary growth models
def heaps_law(N, k=20, beta=0.6):
    return k * (N ** beta)

def lstm_vocab_growth(N):
    return np.log(N) ** 2

def bert_vocab_growth(N, max_vocab=30000):
    return max_vocab * (1 - np.exp(-N / max_vocab))

def laplace_vocab_growth(N, alpha=1, total_vocab=5000):
    return total_vocab * (1 - np.exp(-N / (alpha * total_vocab)))

def katz_vocab_growth(N, d=0.5, total_vocab=5000):
    return total_vocab * (1 - d * np.exp(-N / (total_vocab)))

def calculate_vocabulary(tokens, N):
    return len(set(tokens[:N]))

# Simulated token counts
N_values = np.logspace(3, 6, num=50, dtype=int)

# Plotting all vocabulary growth models
plt.figure(figsize=(12, 6))

# LNRE (Heap's Law)
lnre_vocab = [heaps_law(N) for N in N_values]
plt.plot(N_values, lnre_vocab, label="LNRE (Heap's Law)", color='black', linestyle='solid', linewidth=2)

# LSTM Vocabulary Growth
lstm_vocab = [lstm_vocab_growth(N) for N in N_values]
plt.plot(N_values, lstm_vocab, label="LSTM", color='black', linestyle='dashed', linewidth=2)

# BERT Vocabulary Growth
bert_vocab = [bert_vocab_growth(N) for N in N_values]
plt.plot(N_values, bert_vocab, label="BERT", color='black', linestyle='dotted', linewidth=2)

# Laplace Smoothing Vocabulary Growth
laplace_vocab = [laplace_vocab_growth(N) for N in N_values]
plt.plot(N_values, laplace_vocab, label="Laplace Smoothing", color='black', linestyle='dashdot', linewidth=2)

# Katz Backoff Vocabulary Growth
katz_vocab = [katz_vocab_growth(N) for N in N_values]
plt.plot(N_values, katz_vocab, label="Katz Backoff", color='black', linestyle=(0, (5, 10)), linewidth=2)

# Adding actual vocabulary sizes
actual_vocab_sizes = [calculate_vocabulary([], N) for N in N_values]  # Replace with actual token data
plt.plot(N_values, actual_vocab_sizes, label="Dataset (Actual)", color='black', linestyle=(0, (3, 5, 1, 5)), linewidth=2)

# Log scale for better visualization
plt.xscale("log")
plt.yscale("log")

# Adding labels and legend
plt.xlabel("Number of Tokens (N)", fontsize=12)
plt.ylabel("Vocabulary Size (V)", fontsize=12)
plt.title("Vocabulary Growth Comparison: All Models", fontsize=14)
```

```
plt.legend(fontsize=10, loc='upper left')
plt.grid(True, which="both", linestyle="--", linewidth=0.5)
plt.tight_layout()

# Save and show the plot
plt.savefig("vocabulary_growth_comparison_bw_final.png", dpi=300)
plt.show()

from collections import Counter

# Generate token frequencies from the dataset
# Ensure train_df["text_prepro"] contains tokenized or cleaned text
token_frequencies = Counter()

# Split preprocessed text into tokens and count frequencies
train_df["text_prepro"].dropna().apply(lambda x: token_frequencies.update(x.split()))

# Now you can use token_frequencies in your function
rare_words = get_rare_words(token_frequencies, threshold=5)
print("Rare Words:", rare_words)


import numpy as np
from scipy.integrate import quad

# Define the G-function (e.g., from token frequencies)
def G_function(z, token_frequencies):
    """G-function for a given z and token frequencies."""
    return sum(freq for token, freq in token_frequencies.items() if freq > z)

# Define the Q-function
def Q_function(z, token_frequencies):
    """Q-function as the integral of x * G_function(x)."""
    def integrand(x):
        return x * G_function(x, token_frequencies)

    Q_value, _ = quad(integrand, 0, z)  # Integrate from 0 to z
    return Q_value

# Function to extract rare words
def get_rare_words(token_frequencies, threshold=5):
    """Identify rare words based on a frequency threshold."""
    return {token: freq for token, freq in token_frequencies.items() if freq <= threshold}

# Example use case
rare_words = get_rare_words(token_frequencies, threshold=5)
print("Rare Words:", rare_words)

# Simulate Q-function over time (e.g., for different time windows)
def analyze_trends_over_time(token_frequencies_time_series, z_values):
    """Analyze changes in Q-function over time."""
    Q_values_over_time = []

    for time_step, freq_distribution in enumerate(token_frequencies_time_series):
        Q_values = [Q_function(z, freq_distribution) for z in z_values]
        Q_values_over_time.append(Q_values)

    return Q_values_over_time

import matplotlib.pyplot as plt

def visualize_Q_function(z_values, Q_values, title="Q-Function Visualization"):
    """Plot Q-function values."""
    plt.figure(figsize=(10, 6))
    plt.plot(z_values, Q_values, marker="o", label="Q-function", color = "black")
    plt.xlabel("z")
    plt.ylabel("Q(z)")
    plt.title(title)
```

```
    plt.grid(True)
    plt.legend()
    plt.show()

# Example token frequencies for one dataset
token_frequencies = {
    "word1": 100, "word2": 50, "word3": 5, "word4": 1, "word5": 2
}

# Calculate Q-function for a range of z-values
z_values = np.linspace(1, 50, 10)
Q_values = [Q_function(z, token_frequencies) for z in z_values]

# Visualize Q-function
visualize_Q_function(z_values, Q_values, title="Q-Function Example")

# Track rare words
rare_words = get_rare_words(token_frequencies, threshold=5)
print("Rare Words:", rare_words)

# Example: Analyze trends with synthetic time series data
time_series = [
    {"word1": 100, "word2": 50, "word3": 5, "word4": 1},  # Time step 1
    {"word1": 90, "word2": 40, "word3": 6, "word4": 2},   # Time step 2
    {"word1": 80, "word2": 30, "word3": 8, "word4": 3}    # Time step 3
]
Q_trends = analyze_trends_over_time(time_series, z_values)

# Visualize trends
for i, Q in enumerate(Q_trends):
    visualize_Q_function(z_values, Q, title=f"Q-Function at Time Step {i + 1}")


import numpy as np
import matplotlib.pyplot as plt

# Define G-function
def G_function(z, token_frequencies):
    """G-function as the sum of probabilities greater than z."""
    return sum(freq for token, freq in token_frequencies.items() if freq > z)

# Define Q-function
def Q_function(z, token_frequencies):
    """Q-function as the cumulative contribution of probabilities below z."""
    return sum(freq for token, freq in token_frequencies.items() if freq <= z)


# Generate z-values (thresholds)
z_values = np.linspace(1, max(token_frequencies.values()), 100)

# Compute Q-function for each z
Q_values = [Q_function(z, token_frequencies) for z in z_values]

# Plot the Q-function
plt.figure(figsize=(10, 6))
plt.plot(z_values, Q_values, marker="o", color = "black", label="Q-function")
plt.xlabel("Threshold (z)")
plt.ylabel("Q(z)")
plt.title("Q-Function Trends Across Thresholds")
plt.grid(True)
plt.legend()
plt.savefig("Q-Function Trends Across Thresholds.png")
plt.show()

time_series = [
    {"word1": 100, "word2": 50, "word3": 5},  # Time step 1
    {"word1": 90, "word2": 40, "word3": 8},   # Time step 2
    {"word1": 80, "word2": 30, "word3": 12}   # Time step 3
]
```

```
# Track Q-function over time
z = 10  # Fixed threshold
Q_trends = [Q_function(z, freq_distribution) for freq_distribution in time_series]

# Plot the Q-function trend over time
plt.figure(figsize=(10, 6))
plt.plot(range(len(Q_trends)), Q_trends, marker="o", color = "black",label=f"Q(z={z}) over time")
plt.xlabel("Time Steps")
plt.ylabel("Q(z)")
plt.title("Q-Function Trends Over Time")
plt.grid(True)
plt.legend()
plt.savefig("Q-Function Trends Across Time.png")
plt.show()


import matplotlib.pyplot as plt

# Example Q_function for demonstration (replace with your actual implementation)
def Q_function(z, freq_distribution):
    return sum(f**z for f in freq_distribution) / len(freq_distribution)

# Example time series data (replace with your actual data)
time_series = [
    [1, 2, 3],  # Frequency distribution at time step 1
    [2, 3, 4],  # Frequency distribution at time step 2
    [3, 4, 5],  # And so on
    [4, 5, 6],
    [5, 6, 7],
]

# Plot Q-functions for multiple z-values with distinct line styles
plt.figure(figsize=(10, 6))
line_styles = ['solid', 'dashed', 'dotted']  # Line styles for each z-value
markers = ['o', 's', 'd']  # Markers for each z-value

for i, z in enumerate([5, 10, 20]):
    Q_values = [Q_function(z, freq_distribution) for freq_distribution in time_series]
    plt.plot(
        range(len(Q_values)),
        Q_values,
        linestyle=line_styles[i],
        marker=markers[i],
        label=f"Q(z={z})",
        color='black',  # Ensure black-and-white compliance
        linewidth=2
    )

# Add labels, grid, and legend
plt.xlabel("Time Steps", fontsize=12)
plt.ylabel("Q(z)", fontsize=12)
plt.title("Q-Function Trends for Multiple Thresholds", fontsize=14)
plt.grid(True, linestyle="--", linewidth=0.5)
plt.legend(fontsize=10, loc='upper left')

# Save and show the plot
plt.tight_layout()
plt.savefig("Q-Function_Trends_Across_Multiple_Thresholds_BW.png", dpi=300)
plt.show()

import matplotlib.pyplot as plt

# Example Q_function for demonstration (replace with your actual implementation)
def Q_function(z, freq_distribution):
    return sum(f**z for f in freq_distribution) / len(freq_distribution)

# Example time series data (replace with your actual data)
```

```
time_series = [
    [1, 2, 3],  # Frequency distribution at time step 1
    [2, 3, 4],  # Frequency distribution at time step 2
    [3, 4, 5],  # And so on
    [4, 5, 6],
    [5, 6, 7],
]

# Plot Q-functions for multiple z-values with distinct line styles
plt.figure(figsize=(10, 6))
line_styles = ['solid', 'dashed', 'dotted']  # Line styles for each z-value
markers = ['o', 's', 'd']  # Markers for each z-value

for i, z in enumerate([5, 10, 20]):
    Q_values = [Q_function(z, freq_distribution) for freq_distribution in time_series]
    plt.plot(
        range(len(Q_values)),
        Q_values,
        linestyle=line_styles[i],
        marker=markers[i],
        label=f"Q(z={z})",
        color='black',  # Ensure black-and-white compliance
        linewidth=2
    )

# Add labels, grid, and legend
plt.xlabel("Time Steps", fontsize=12)
plt.ylabel("Q(z)", fontsize=12)
plt.title("Q-Function Trends for Multiple Thresholds", fontsize=14)
plt.grid(True, linestyle="--", linewidth=0.5)
plt.legend(fontsize=10, loc='upper left')

# Save and show the plot
plt.tight_layout()
plt.savefig("Q-Function_Trends_Across_Multiple_Thresholds_BW.png", dpi=300)
plt.show()


from mpl_toolkits.mplot3d import Axes3D
import numpy as np
import matplotlib.pyplot as plt

# Define Q-function
def Q_function(z, token_frequencies):
    """Q-function as the cumulative contribution of probabilities below z."""
    return sum(freq for token, freq in token_frequencies.items() if freq <= z)

# Simulated time-series data (replace with real data)
time_series = [
    {"word1": 100, "word2": 50, "word3": 5, "word4": 1},
    {"word1": 90, "word2": 40, "word3": 8, "word4": 2},
    {"word1": 80, "word2": 30, "word3": 12, "word4": 4},
]

# Generate z-values and compute Q-function over time
z_values = np.linspace(1, 100, 50)
Q_values_over_time = [
    [Q_function(z, freq_distribution) for z in z_values] for freq_distribution in time_series
]

# Create 3D plot
fig = plt.figure(figsize=(10, 6))
ax = fig.add_subplot(111, projection='3d')

time_steps = np.arange(len(time_series))
Z, T = np.meshgrid(z_values, time_steps)
Q = np.array(Q_values_over_time)

```

```python
# Use wireframe instead of a colored surface for black-and-white compatibility
ax.plot_wireframe(Z, T, Q, color='black', linewidth=0.8)

# Add labels and title
ax.set_xlabel("Threshold (z)", fontsize=10)
ax.set_ylabel("Time Steps", fontsize=10)
ax.set_zlabel("Q(z)", fontsize=10)
ax.set_title("3D Q-Function Trends Over Time", fontsize=12)

# Save and display the plot
plt.tight_layout()
plt.savefig("3D_Q_Function_Trends_BW.png", dpi=300, bbox_inches='tight')
plt.show()



import plotly.graph_objects as go
import numpy as np

# Define Q-function
def Q_function(z, token_frequencies):
    """Q-function as the cumulative contribution of probabilities below z."""
    return sum(freq for token, freq in token_frequencies.items() if freq <= z)

# Simulated time-series data (replace with real data)
time_series = [
    {"word1": 100, "word2": 50, "word3": 5, "word4": 1},
    {"word1": 90, "word2": 40, "word3": 8, "word4": 2},
    {"word1": 80, "word2": 30, "word3": 12, "word4": 4},
]

# Generate Q-function values
z_values = np.linspace(1, 100, 50)
time_steps = np.arange(len(time_series))
Q_values_over_time = [
    [Q_function(z, freq_distribution) for z in z_values] for freq_distribution in
    time_series
]

# Create a static plot with distinct dash patterns
fig = go.Figure()

dash_styles = ["solid", "dash", "dot"]  # Different line styles for time steps

for t, (Q_vals, dash_style) in enumerate(zip(Q_values_over_time, dash_styles)):
    fig.add_trace(go.Scatter(
        x=z_values, y=Q_vals,
        mode='lines',
        line=dict(dash=dash_style, color='black', width=2),
        name=f"Time Step {t+1}"
    ))

fig.update_layout(
    title="Static Q-Function Visualization for Different Time Steps",
    xaxis_title="Threshold (z)",
    yaxis_title="Q(z)",
    legend_title="Time Step",
    template="plotly_white",
    font=dict(size=12),
)

# Save the figure as an image (optional)
fig.write_image("Static_Q_Function_Trends_BW.png", width=800, height=600)

# Show the plot
fig.show()
```

```python
from matplotlib.animation import FuncAnimation

# Create figure
fig, ax = plt.subplots(figsize=(10, 6))
line, = ax.plot([], [], label="Q-function", color = "black")
ax.set_xlim(1, max(token_frequencies.values()))
ax.set_ylim(0, max(Q_values_over_time[-1]))
ax.set_xlabel("Threshold (z)")
ax.set_ylabel("Q(z)")
ax.set_title("Animated Q-Function Trends")
ax.grid(True)
ax.legend()

# Update function for animation
def animate(frame):
    Q_vals = Q_values_over_time[frame]
    line.set_data(z_values, Q_vals)
    ax.set_title(f"Time Step {frame + 1}")
    return line,

# Create animation
ani = FuncAnimation(fig, animate, frames=len(time_series), interval=1000, blit=True)
plt.show()


import numpy as np
import matplotlib.pyplot as plt

# Sample Q-function values for different thresholds z
thresholds = np.linspace(1, 100, 100)
Q_values = [np.log10(z) * z if z < 50 else z ** 0.8 for z in thresholds]

# Log-Log Plot
plt.figure(figsize=(10, 6))
plt.plot(thresholds, Q_values, label="Q-function", marker='o', linestyle='-',
color='black')
plt.xscale("log")
plt.yscale("log")
plt.xlabel("Threshold (z) (Log Scale)", fontsize=12)
plt.ylabel("Q(z) (Log Scale)", fontsize=12)
plt.title("Log-Log Plot of Q-Function", fontsize=14)
plt.grid(True, which="both", linestyle="--", linewidth=0.5)
plt.legend()
plt.tight_layout()
plt.savefig("Log-Log Plot of Q-function.png")
plt.show()

import numpy as np

# Step 1: Compute token frequencies
token_frequencies = {"word1": 3, "word2": 7, "word3": 1, "word4": 5, "word5": 2}

# Step 2: Define thresholds
thresholds = range(1, 10)

# Step 3: Compute Q-function values
def compute_q_function(token_frequencies, thresholds):
    q_values = []
    for z in thresholds:
        q_value = sum(freq for freq in token_frequencies.values() if freq <= z)
        q_values.append(q_value)
    return q_values

q_values = compute_q_function(token_frequencies, thresholds)

# Step 4: Identify rare events
def identify_rare_events(data, q_function_values, threshold=5):
    rare_indices = [i for i, q_val in enumerate(q_function_values) if q_val < threshold]
    rare_events = [data[i] for i in rare_indices]
```

```
    return rare_events

rare_events = identify_rare_events(list(token_frequencies.keys()), q_values, threshold=5)

print("Rare Events:", rare_events)


import numpy as np
import matplotlib.pyplot as plt

# Simulated time-series Q-function values for different thresholds
time_steps = np.arange(0, 3, 0.5)  # Example time steps
thresholds = [10, 30, 50]  # Specific thresholds to analyze
Q_time_series = {
    10: [10, 12, 15, 10, 5, 0],  # Q(z=10) over time
    30: [20, 25, 30, 40, 50, 55],  # Q(z=30) over time
    50: [30, 35, 40, 60, 80, 100]  # Q(z=50) over time
}

# Define line styles for each threshold
line_styles = ['solid', 'dotted', 'dashdot']  # Different line styles for thresholds
markers = ['o', 's', 'd']  # Different markers for thresholds

# Plot Temporal Trends
plt.figure(figsize=(10, 6))

for (z, values), line_style, marker in zip(Q_time_series.items(), line_styles, markers):
    plt.plot(
        time_steps,
        values,
        label=f"Q(z={z})",
        marker=marker,
        linestyle=line_style,
        color='black',  # Ensure black-and-white compliance
        linewidth=2,
        markersize=6,
    )

# Add labels, title, legend, and grid
plt.xlabel("Time Steps", fontsize=12)
plt.ylabel("Q(z)", fontsize=12)
plt.title("Temporal Trends of Q-Function for Specific Thresholds", fontsize=14)
plt.grid(True, linestyle="--", linewidth=0.5)
plt.legend(fontsize=10, loc='upper left')

# Save and display the plot
plt.tight_layout()
plt.savefig("Temporal_Trends_of_Q_Function_BW.png", dpi=300)
plt.show()


# Simulated Q-function and G-function values
thresholds = np.linspace(1, 100, 100)
Q_values = [np.log10(z) * z if z < 50 else z ** 0.8 for z in thresholds]
G_values = [z ** 0.5 for z in thresholds]  # Example G-function trend

# Overlay Plot
plt.figure(figsize=(10, 6))
plt.plot(thresholds, Q_values, label="Q-function", color='black', linestyle =
"dotted",linewidth=2)
plt.plot(thresholds, G_values, label="G-function", color='black', linestyle="--",
linewidth=2)
plt.xscale("log")
plt.yscale("log")
plt.xlabel("Threshold (z) (Log Scale)", fontsize=12)
plt.ylabel("Value (Log Scale)", fontsize=12)
plt.title("Comparison of Q-Function and G-Function Trends", fontsize=14)
plt.grid(True, which="both", linestyle="--", linewidth=0.5)
plt.legend()
```

```
plt.tight_layout()
plt.savefig("Q-function vs Q-function trends.png")
plt.show()


def map_rare_events_to_context(data, rare_events):
    """
    Map rare events back to their original context in the dataset.

    Parameters:
    - data: List of original texts (e.g., user reviews or feedback).
    - rare_events: List of rare tokens.

    Returns:
    - rare_event_contexts: Dictionary mapping rare tokens to their contexts.
    """
    rare_event_contexts = {event: [] for event in rare_events}
    for event in rare_events:
        for text in data:
            if event in text:
                rare_event_contexts[event].append(text)
    return rare_event_contexts

# Example usage
rare_event_contexts = map_rare_events_to_context(train_df["text_prepro"], rare_events)
for event, contexts in rare_event_contexts.items():
    print(f"Rare Event: {event}")
    print("Contexts:", contexts)



def visualize_rare_token_q_contributions(rare_events, q_values):
    """
    Visualize Q-function contributions for rare events.

    Parameters:
    - rare_events: List of rare tokens.
    - q_values: Q-function values corresponding to thresholds.
    """
    rare_q_values = [q_values[i] for i, token in enumerate(rare_events)]
    plt.figure(figsize=(12, 6))
    plt.bar(rare_events, rare_q_values, color="black")
    plt.xlabel("Rare Tokens")
    plt.ylabel("Q(z)")
    plt.title("Q-Function Contributions for Rare Tokens")
    plt.xticks(rotation=45)
    plt.grid(True)
    plt.show()

# Example usage
visualize_rare_token_q_contributions(rare_events, q_values)



def extract_contexts(text_data, rare_event, window=3):
    """
    Extract contexts for a rare event within a given window size.

    Parameters:
    - text_data: List of tokenized texts (sentences or documents).
    - rare_event: The rare event (word) to find contexts for.
    - window: Number of words before and after the rare event to include in the context.

    Returns:
    - contexts: List of contexts (substrings or word windows) where the rare event occurs.
    """
    contexts = []
    for text in text_data:
```

```
        words = text.split()  # Assuming `text_data` is tokenized
        for i, word in enumerate(words):
            if word == rare_event:
                start = max(0, i - window)
                end = min(len(words), i + window + 1)
                contexts.append(" ".join(words[start:end]))
    return contexts



# Example text data
text_data = ["word1 is an example", "word2 appears here", "word1 and word3 are rare"]

# Extract contexts for a rare event
rare_event = "word1"
contexts = extract_contexts(text_data, rare_event, window=2)
print("Contexts for", rare_event, ":", contexts)


import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import poisson

# Generate synthetic data for demonstration
np.random.seed(42)
frequencies = np.random.poisson(lam=5, size=1000)  # Simulated token frequencies

# Define empirical G-function (cumulative contribution above threshold z)
def empirical_g_function(frequencies, z):
    """Compute the empirical G-function for threshold z."""
    return sum(freq for freq in frequencies if freq > z)

# Define theoretical C-function
def theoretical_c_function(z, Q_values):
    """
    Compute the theoretical C-function based on the Q-function.
    Q_values should represent cumulative contributions (e.g., from rare events).
    """
    # Approximation using summation for Q-function values
    C = sum((poisson.sf(k=z, mu=q) * q for q in Q_values))
    return C

# Simulate Q-function values (cumulative contributions from rare events)
Q_values = np.linspace(0.1, 50, 100)  # Rare-event contributions

# Define thresholds
thresholds = np.linspace(0, max(frequencies), 50)

# Compute empirical G-function and theoretical C-function for each threshold
empirical_g = [empirical_g_function(frequencies, z) for z in thresholds]
theoretical_c = [theoretical_c_function(z, Q_values) for z in thresholds]

# Visualization of Convergence
plt.figure(figsize=(12, 6))
plt.plot(thresholds, empirical_g, label="Empirical G-function", marker="o", linestyle="-", color = "black")
plt.plot(thresholds, theoretical_c, label="Theoretical C-function", marker="x", linestyle="--", color = "black")
plt.xlabel("Threshold (z)", fontsize=12)
plt.ylabel("Cumulative Contribution", fontsize=12)
plt.title("Convergence of G-function to C-function", fontsize=14)
plt.legend(fontsize=12)
plt.grid(True)
plt.tight_layout()
plt.savefig("convergence of G-function to C-function.png")
plt.show()

# Analyze the difference (convergence behavior)
convergence_difference = np.abs(np.array(empirical_g) - np.array(theoretical_c))
```

```
# Visualize the difference
plt.figure(figsize=(12, 6))
plt.plot(thresholds, convergence_difference, label="Difference (|Empirical G - Theoretical C|)", color="black")
plt.xlabel("Threshold (z)", fontsize=12)
plt.ylabel("Difference", fontsize=12)
plt.title("Convergence Difference Between Empirical G and Theoretical C", fontsize=14)
plt.legend(fontsize=12)
plt.grid(True)
plt.tight_layout()
plt.savefig("convergence difference between G-function.png")
plt.show()



import numpy as np
import matplotlib.pyplot as plt

# Simulated data for empirical G-function and theoretical C-function
z_values = np.arange(0, 12, 1)  # Threshold values (z)
empirical_G_values = np.random.randint(4000, 5000, len(z_values))  # Simulated empirical G-function
theoretical_C_values = np.full(len(z_values), 3000)  # Simulated theoretical C-function as constant

# Model Calibration Function
def calibrate_model(empirical_G, theoretical_C, weights=None):
    """
    Calibrate the empirical G-function to better align with the theoretical C-function.
    """
    if weights is None:
        weights = np.ones_like(empirical_G)  # Default equal weighting

    calibration_factor = (empirical_G - theoretical_C) * weights
    calibrated_G = empirical_G - calibration_factor

    return calibrated_G

# Apply model calibration
weights = np.linspace(1, 0.1, len(z_values))  # Example: higher weight for lower thresholds
calibrated_G = calibrate_model(empirical_G_values, theoretical_C_values, weights=weights)

# Plot calibrated G-function against theoretical C-function
plt.figure(figsize=(10, 5))
plt.plot(z_values, calibrated_G, label="Calibrated G-function", color="black", linestyle = "dotted")
plt.plot(z_values, theoretical_C_values, label="Theoretical C-function", color="black", linestyle="-.")
plt.xlabel("Threshold (z)")
plt.ylabel("Cumulative Contribution")
plt.title("Calibrated G-function vs Theoretical C-function")
plt.legend()
plt.grid(True)
plt.savefig("calibrated G-functions vs Theoretical C-function.png")
plt.show()



import numpy as np
import matplotlib.pyplot as plt

# Simulate temporal evolution of G_f(z) and C(z)
time_steps = np.arange(1, 11)  # Example: 10 time steps
z_values = np.linspace(1, 50, 50)  # Threshold values (example)
empirical_G_values = np.exp(-z_values / 10)  # Simulated initial G-values
theoretical_C_values = np.exp(-z_values / 20)  # Simulated baseline C-values

```

```
# Simulated growth over time
temporal_G_values = [empirical_G_values * (1 + 0.1 * t) for t in time_steps]  # Simulated G-function growth
temporal_C_values = [theoretical_C_values * (1 + 0.05 * t) for t in time_steps]  # Simulated C-function growth

# Visualization
plt.figure(figsize=(10, 5))

# Define line styles and markers
line_styles = ['solid', 'dotted', 'dashdot', 'dashed']
markers = ['o', 's', 'd', 'x']

# Plot temporal G-function trends
for t, G_values in enumerate(temporal_G_values, start=1):
    linestyle = line_styles[t % len(line_styles)]  # Cycle through line styles
    marker = markers[t % len(markers)]  # Cycle through markers
    plt.plot(
        z_values,
        G_values,
        label=f"G-function (t={t})",
        linestyle=linestyle,
        marker=marker,
        color='black',
        linewidth=1.5,
        markersize=5,
    )

# Plot baseline theoretical C-function
plt.plot(
    z_values,
    theoretical_C_values,
    label="Theoretical C-function (Baseline)",
    color='black',
    linestyle='--',
    linewidth=2,
)

# Add labels, title, legend, and grid
plt.xlabel("Threshold (z)", fontsize=12)
plt.ylabel("Cumulative Contribution", fontsize=12)
plt.title("Temporal Evolution of G-function", fontsize=14)
plt.grid(True, linestyle="--", linewidth=0.5)
plt.legend(fontsize=10, loc='upper left')

# Save and display the plot
plt.tight_layout()
plt.savefig("temporal_evolution_of_G_function_BW.png", dpi=300)
plt.show()



import numpy as np
import matplotlib.pyplot as plt

# Threshold values (z)
z_values = np.arange(0, 12, 1)

# Simulated data for empirical G-function for multiple datasets
dataset1_G = np.random.randint(4000, 5000, len(z_values))  # Simulated dataset 1
dataset2_G = np.random.randint(3000, 4500, len(z_values))  # Simulated dataset 2
dataset3_G = np.random.randint(2000, 4000, len(z_values))  # Simulated dataset 3

# Prepare datasets for plotting
datasets = {
    "Dataset 1": dataset1_G,
    "Dataset 2": dataset2_G,
    "Dataset 3": dataset3_G,
}
```

```python

# Define line styles and markers for distinction
line_styles = ["solid", "dashed", "dotted"]
markers = ["o", "s", "d"]

# Plot empirical G-functions for each dataset
plt.figure(figsize=(10, 6))

for (name, G_values), linestyle, marker in zip(datasets.items(), line_styles, markers):
    plt.plot(
        z_values,
        G_values,
        label=name,
        linestyle=linestyle,
        marker=marker,
        color="black",  # Use black for all lines
        linewidth=1.5,
        markersize=6,
    )

# Plot theoretical C-function
theoretical_C_values = np.full(len(z_values), 3000)  # Simulated theoretical C-function as constant
plt.plot(
    z_values,
    theoretical_C_values,
    label="Theoretical C-function",
    linestyle="dashdot",  # Unique style for theoretical C-function
    color="black",
    linewidth=2,
)

# Add labels, legend, and grid
plt.xlabel("Threshold (z)", fontsize=12)
plt.ylabel("Cumulative Contribution", fontsize=12)
plt.title("Comparison of Empirical G-function Across Datasets", fontsize=14)
plt.legend(fontsize=10, loc="upper right")
plt.grid(True, linestyle="--", linewidth=0.5)

# Save and display the plot
plt.tight_layout()
plt.savefig("Comparison_of_Empirical_G_Function_BW.png", dpi=300)
plt.show()


# Identify changes in rare event contributions
def identify_trends(temporal_G_values, threshold_z):
    trends = []
    for t, G_values in enumerate(temporal_G_values):
        contribution = G_values[threshold_z]
        trends.append((t, contribution))
    return trends

trends = identify_trends(temporal_G_values, threshold_z=5)
print("Trends in rare event contributions over time:", trends)



import numpy as np
import matplotlib.pyplot as plt
from collections import Counter

# Example: Token frequency calculation
tokens = [word for text in train_df["text_prepro"] for word in text.split()]
token_counts = Counter(tokens)

# Sort by frequency
sorted_token_counts = sorted(token_counts.values(), reverse=True)

```

```
# Plot frequency distribution
plt.figure(figsize=(10, 6))
plt.loglog(range(1, len(sorted_token_counts) + 1), sorted_token_counts, color =
"black",marker='o', linestyle='-')
plt.title("Log-Log Plot of Token Frequency Distribution")
plt.xlabel("Rank (log scale)")
plt.ylabel("Frequency (log scale)")
plt.grid(True, which="both", linestyle="--", linewidth=0.5)
plt.savefig("Log-Log Plot of Token Frequency Distribution.png")
plt.show()


import numpy as np
import matplotlib.pyplot as plt

# Simulated sorted token counts (replace with your actual data)
sorted_token_counts = np.random.zipf(2, 100)  # Simulated token frequencies

# Rank-Frequency Product
ranks = np.arange(1, len(sorted_token_counts) + 1)
frequencies = np.array(sorted_token_counts)
rank_freq_product = ranks * frequencies

# Plot observed vs. theoretical
plt.figure(figsize=(10, 6))

# Observed data
plt.loglog(
    ranks,
    frequencies,
    label="Observed",
    linestyle="solid",
    color="black",
    linewidth=1.5,
)

# Theoretical curve
plt.loglog(
    ranks,
    1 / ranks,
    label="Theoretical (1/r)",
    linestyle="dashed",
    color="black",
    linewidth=1.5,
)

# Add title, labels, legend, and grid
plt.title("Validation of Zipf's Law", fontsize=14)
plt.xlabel("Rank (log scale)", fontsize=12)
plt.ylabel("Frequency (log scale)", fontsize=12)
plt.legend(fontsize=10, loc="upper right")
plt.grid(True, which="both", linestyle="--", linewidth=0.5)

# Save and display the plot
plt.tight_layout()
plt.savefig("Validation_of_Zipfs_Law_BW.png", dpi=300)
plt.show()



# Cumulative Distribution Function
cumulative_frequencies = np.cumsum(frequencies) / sum(frequencies)

# Plot CDF
plt.figure(figsize=(10, 6))
plt.plot(ranks, cumulative_frequencies, color = "black", linestyle =
"solid",label="Cumulative Frequency")
plt.axhline(0.9, color='black', linestyle='--', label="90% Threshold")
plt.title("Cumulative Distribution Function")
```

```python
plt.xlabel("Rank")
plt.ylabel("Cumulative Frequency")
plt.legend()
plt.grid(True)
plt.savefig("Cumulative Distribution Function.png")
plt.show()

# Percentage Contribution of Rare Events (e.g., bottom 10%)
tail_threshold = int(len(sorted_token_counts) * 0.9)  # Bottom 10% ranks
tail_contribution = cumulative_frequencies[tail_threshold]
print(f"Contribution of the rare events (tail): {tail_contribution:.2%}")



from scipy.stats import powerlaw

# Fit Power-Law Distribution
a, loc, scale = powerlaw.fit(frequencies, floc=0)
theoretical_freq = powerlaw.pdf(ranks, a, loc, scale)

# Plot Observed vs. Fitted
plt.figure(figsize=(10, 6))
plt.loglog(ranks, frequencies, label="Observed", color = "black")
plt.loglog(ranks, theoretical_freq, label="Fitted Power-Law", color = "black", linestyle='--')
plt.title("Power-Law Fit to Frequency Distribution")
plt.xlabel("Rank (log scale)")
plt.ylabel("Frequency (log scale)")
plt.legend()
plt.grid(True, which="both", linestyle="--", linewidth=0.5)
plt.savefig("Power-Law Fit to Frequency Distribution.png")


from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.cluster import KMeans

# --- Semantic Clustering: TF-IDF and KMeans Clustering ---
# Prepare the data for clustering
vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(rare_words_df["Word"])

# Apply KMeans clustering
num_clusters = 5  # Specify the number of clusters
kmeans = KMeans(n_clusters=num_clusters, random_state=42)
kmeans.fit(X)

# Add cluster labels to the rare_words_df
rare_words_df["Cluster"] = kmeans.labels_

# Visualize the clusters
plt.figure(figsize=(10, 6))
for cluster_id in range(num_clusters):
    cluster_words = rare_words_df[rare_words_df["Cluster"] == cluster_id]["Word"]
    plt.bar(cluster_words, [1] * len(cluster_words), label=f"Cluster {cluster_id}")
plt.xticks(rotation=90)
plt.xlabel("Words")
plt.ylabel("Cluster Indicator")
plt.title("Rare Word Clustering")
plt.legend()
plt.tight_layout()
plt.show()



import matplotlib.pyplot as plt
from wordcloud import WordCloud

# Word cloud for each cluster
for cluster_id in range(num_clusters):
```

```python
    cluster_words = rare_words_df[rare_words_df["Cluster"] == cluster_id]
    word_freq = dict(zip(cluster_words["Word"], cluster_words["Frequency"]))

    # Generate word cloud
    wordcloud = WordCloud(
        width=800, height=400, background_color='white'
    ).generate_from_frequencies(word_freq)

    # Plot the word cloud
    plt.figure(figsize=(10, 5))
    plt.imshow(wordcloud, interpolation='bilinear')
    plt.title(f"Word Cloud for Cluster {cluster_id}")
    plt.savefig("wordcloud.png")
    plt.axis("off")
    plt.show()



import numpy as np
import matplotlib.pyplot as plt

# Example data: Replace this with actual frequency and rank data
ranks = np.arange(1, 10001)  # Ranks
frequencies = 1 / (ranks ** 1.2)  # Example power-law distribution

plt.figure(figsize=(10, 6))
plt.loglog(ranks, frequencies, marker="o", linestyle="none", color = "black", label="Observed Data")
plt.xlabel("Rank (log scale)")
plt.ylabel("Frequency (log scale)")
plt.title("Log-Log Plot of Token Frequency Distribution")
plt.legend()
plt.grid(True, which="both", linestyle="--")
plt.savefig("Log-Log Plot of Token Frequency Distribution.png")
plt.show()



# Example G_f(z) and C(z) values (replace with actual calculations)
z_values = np.arange(1, 20)  # Thresholds
empirical_G = 5000 / z_values  # Replace with actual G_f(z) computation
theoretical_C = 3000 / z_values  # Replace with actual C(z)

difference = np.abs(empirical_G - theoretical_C)

plt.figure(figsize=(10, 6))
plt.plot(z_values, empirical_G, label="Empirical G_f(z)", color="black")
plt.plot(z_values, theoretical_C, label="Theoretical C(z)", linestyle="--", color="black")
plt.plot(z_values, difference, label="Difference", linestyle=":", color="black")
plt.xlabel("Threshold (z)")
plt.ylabel("Cumulative Contribution")
plt.title("Convergence Analysis: Empirical G_f(z) vs Theoretical C(z)")
plt.legend()
plt.grid()
plt.savefig("Convergence Analysis: Empirical G_f(z) vs Theoretical C(z).png")
plt.show()



from powerlaw import Fit

# Example data: Replace with actual token frequencies
data = np.random.zipf(a=1.5, size=1000)  # Example Zipf distribution
fit = Fit(data)

print(f"Alpha (Scaling Parameter): {fit.alpha}")
print(f"KS Test Statistic: {fit.D}")
print(f"P-value: {fit.power_law.D}")
```

```
# Variance of contributions from rare events
rare_event_contributions = empirical_G - theoretical_C
variance = np.var(rare_event_contributions)

print(f"Variance of Rare Event Contributions: {variance}")


import numpy as np
import matplotlib.pyplot as plt

# Simulate data
z_values = np.linspace(1, 50, 50)  # Threshold values
time_steps = np.arange(1, 11)  # Example time steps
empirical_G = np.exp(-z_values / 10)  # Simulated empirical G-function
rare_event_contributions_over_time = [
    empirical_G / t for t in time_steps  # Simulate decreasing contributions over time
]

# Define line styles and markers for distinction
line_styles = ["solid", "dashed", "dotted", "dashdot"]
markers = ["o", "s", "d", "x"]

# Create plot
plt.figure(figsize=(10, 6))

for t, contributions in zip(time_steps, rare_event_contributions_over_time):
    linestyle = line_styles[t % len(line_styles)]  # Cycle through line styles
    marker = markers[t % len(markers)]  # Cycle through markers
    plt.plot(
        z_values,
        contributions,
        label=f"Time Step {t}",
        linestyle=linestyle,
        marker=marker,
        color="black",  # Black for all lines
        linewidth=1.5,
        markersize=5,
    )

# Add labels, title, legend, and grid
plt.xlabel("Threshold (z)", fontsize=12)
plt.ylabel("Rare Event Contributions", fontsize=12)
plt.title("Temporal Analysis of Rare Event Contributions", fontsize=14)
plt.grid(True, linestyle="--", linewidth=0.5)
plt.legend(fontsize=10, loc="upper right")

# Save and display the plot
plt.tight_layout()
plt.savefig("Temporal_Analysis_of_Rare_Event_Contributions_BW.png", dpi=300)
plt.show()


from scipy.optimize import curve_fit

# Function for fitting a power-law
def power_law(x, a, b):
    return b * (x ** -a)

# Fit power-law to data
params, _ = curve_fit(power_law, ranks, frequencies)
alpha = params[0]
print(f"Tail Index (Alpha): {alpha}")



from numpy.linalg import eigvalsh
```

```
# Example covariance matrix
cov_matrix = np.random.rand(10, 10)
eigenvalues = eigvalsh(cov_matrix)

plt.figure(figsize=(10, 6))
plt.plot(np.sort(eigenvalues), marker="o", label="Eigenvalues")
plt.xlabel("Index")
plt.ylabel("Eigenvalue")
plt.title("Spectral Analysis: Eigenvalues of Covariance Matrix")
plt.legend()
plt.grid()
plt.savefig("Spectral Analysis: Eigenvalues of Covariance Matrix.png")
plt.show()



# Example tail contributions
tail_contributions = np.cumsum(frequencies[::-1]) / np.sum(frequencies)

plt.figure(figsize=(10, 6))
plt.plot(ranks[::-1], tail_contributions, label="Tail Contribution", color = "black",
linestyle = "dotted")
plt.axhline(0.9, color="black", linestyle="--", label="90% Contribution Threshold")
plt.xlabel("Rank (Descending Order)")
plt.ylabel("Cumulative Contribution")
plt.title("Tail Contribution Analysis")
plt.legend()
plt.grid()
plt.savefig("Tail Contribution Analysis.png")
plt.show()



# Define a threshold for residuals (e.g., 1% of average vocabulary size)
a_threshold = 0.01 * np.mean(actual_vocab_sizes)  # 1% of the mean vocabulary size

# Residual difference between observed and LNRE (Heap's Law)
residuals = np.abs(np.array(actual_vocab_sizes) - np.array(lnre_vocab))

# Check convergence
if np.max(residuals) < a_threshold:
    print("Vocabulary growth stabilizes (converges globally).")
else:
    print("Vocabulary growth indicates divergence.")



import numpy as np
import matplotlib.pyplot as plt
from collections import Counter

# ----------------------------- Functions -----------------------------

# Tokenize function (example implementation)
def tokenize(text):
    """Basic tokenizer splitting by whitespace."""
    return text.split()

# Frequency distribution calculation
def calculate_frequency_distribution(tokens):
    """Calculate the frequency distribution of tokens."""
    token_counts = Counter(tokens)
    total_count = sum(token_counts.values())
    return {token: count / total_count for token, count in token_counts.items()}

# Vocabulary calculation
def calculate_vocabulary(tokens, N):
    """Calculate vocabulary size for the first N tokens."""
```

```
    return len(set(tokens[:N]))

# Rare event frequency calculation
def calculate_rare_event_frequencies(tokens, threshold=0.01):
    """
    Identify rare events based on a frequency threshold.

    Parameters:
    - tokens: List of tokens
    - threshold: Frequency threshold for defining rare events

    Returns:
    - List of rare event frequencies
    """
    freq_dist = calculate_frequency_distribution(tokens)
    rare_events = {token: freq for token, freq in freq_dist.items() if freq < threshold}
    return list(rare_events.values())

# Rank calculation (Zipf's law)
def rank(tokens):
    """Calculate rank of tokens based on their frequency."""
    freq_dist = Counter(tokens)
    return sorted(freq_dist.values(), reverse=True)

# Rare event contributions
def calculate_rare_event_contributions(tokens, N, threshold=0.01):
    """
    Calculate the contributions of rare events for the first N tokens.

    Parameters:
    - tokens: List of tokens
    - N: Number of tokens to consider
    - threshold: Frequency threshold for rare events

    Returns:
    - Contribution of rare events
    """
    token_counts = Counter(tokens[:N])
    total_tokens = sum(token_counts.values())
    rare_events = {token: count for token, count in token_counts.items() if count / total_tokens < threshold}
    rare_contribution = sum(rare_events.values()) / total_tokens
    return rare_contribution

# Vocabulary growth models
def heaps_law(N, k=20, beta=0.6):
    """Vocabulary size based on Heap's law (LNRE)."""
    return k * (N ** beta)

def lstm_vocab_growth(N):
    """
    Simulate LSTM vocabulary growth.
    This assumes diminishing returns due to training bias toward frequent words.
    """
    return np.log(N) ** 2

def bert_vocab_growth(N, max_vocab=30000):
    """
    Simulate BERT vocabulary growth.
    BERT relies on subword tokenization, so the vocabulary growth saturates early.
    """
    return max_vocab * (1 - np.exp(-N / max_vocab))

# ----------------------------- Data and Analysis -----------------------------

# Example tokens (replace with actual dataset)
tokens = ["word" + str(i) for i in range(1, 10001)] * 5  # Example dataset with repeated tokens

```

```python
# Token counts for analysis
N_values = np.logspace(3, 6, num=50, dtype=int)  # Token counts from 1,000 to 1,000,000

# Actual vocabulary sizes from the dataset
actual_vocab_sizes = [calculate_vocabulary(tokens, N) for N in N_values]

# Rare event contributions
rare_event_contributions = np.array([calculate_rare_event_contributions(tokens, N) for N in N_values])

# Simulate LNRE (Heap's Law), LSTM, and BERT growth for comparison
lnre_vocab = [heaps_law(N) for N in N_values]
lstm_vocab = [lstm_vocab_growth(N) for N in N_values]
bert_vocab = [bert_vocab_growth(N) for N in N_values]

# ----------------------------- Plots -----------------------------

# 1. Vocabulary Growth Comparison
plt.figure(figsize=(12, 6))
plt.plot(N_values, lnre_vocab, label="LNRE (Heap's Law)", color="blue", linewidth=2)
plt.plot(N_values, lstm_vocab, label="LSTM", color="red", linestyle="--", linewidth=2)
plt.plot(N_values, bert_vocab, label="BERT", color="green", linestyle=":", linewidth=2)
plt.plot(N_values, actual_vocab_sizes, label="Dataset (Actual)", color="purple", linewidth=2)

# Log scale for better visualization
plt.xscale("log")
plt.yscale("log")
plt.xlabel("Number of Tokens (N)", fontsize=12)
plt.ylabel("Vocabulary Size (V)", fontsize=12)
plt.title("Vocabulary Growth Comparison: LNRE vs LSTM vs BERT", fontsize=14)
plt.legend(fontsize=12)
plt.grid(True, which="both", linestyle="--", linewidth=0.5)
plt.tight_layout()
plt.savefig("vocabulary_growth_comparison.png", dpi=300)
plt.show()

# 2. Rare Event Contributions
plt.figure(figsize=(12, 6))
plt.plot(N_values, rare_event_contributions, label="Rare Event Contributions", color="orange", linewidth=2)
plt.xscale("log")
plt.xlabel("Number of Tokens (N)", fontsize=12)
plt.ylabel("Rare Event Contribution", fontsize=12)
plt.title("Rare Event Contributions Over Tokens", fontsize=14)
plt.legend(fontsize=12)
plt.grid(True, which="both", linestyle="--", linewidth=0.5)
plt.tight_layout()
plt.savefig("rare_event_contributions.png", dpi=300)
plt.show()

# ----------------------------- Convergence Analysis -----------------------------

# Residual difference between observed and LNRE (Heap's Law)
residuals = np.abs(np.array(actual_vocab_sizes) - np.array(lnre_vocab))
a_threshold = 0.05  # Example threshold for convergence
if np.max(residuals) < a_threshold:
    print("Vocabulary growth stabilizes (converges globally).")
else:
    print("Vocabulary growth fluctuates (diverges globally).")

# Variance reduction for rare events
variance_reduction = np.var(rare_event_contributions, axis=0)
if np.max(np.abs(np.diff(variance_reduction))) < 1e-5:  # Small threshold for stabilization
    print("Rare event contributions stabilize, indicating global convergence.")
else:
    print("Rare event contributions fluctuate, indicating divergence.")
```

```python
# Variance reduction for rare events
variance_reduction = np.var(rare_event_contributions)  # Variance is a scalar in this case

# Check for stabilization
if variance_reduction < 1e-5:  # Small threshold for stabilization
    print("Rare event contributions stabilize, indicating global convergence.")
else:
    print("Rare event contributions fluctuate, indicating divergence.")



import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from collections import Counter

# Step 1: Extract token frequencies from train_df["text_prepro"]
tokens = [word for text in train_df["text_prepro"].dropna() for word in text.split()]
token_counts = Counter(tokens)

# Step 2: Sort tokens by frequency
sorted_token_counts = sorted(token_counts.values(), reverse=True)
rank = np.arange(1, len(sorted_token_counts) + 1)  # Rank of tokens

# Step 3: Normalize demand distribution and calculate cumulative contribution
demand = np.array(sorted_token_counts) / sum(sorted_token_counts)  # Normalize demands
cumulative_demand = np.cumsum(demand)  # Cumulative contribution

# Step 4: Define the length of the tail
tail_threshold = int(0.9 * len(sorted_token_counts))  # Bottom 90% of tokens
tail_contribution = cumulative_demand[tail_threshold - 1]  # Contribution from the tail

# Step 5: Plot the conceptual model
plt.figure(figsize=(12, 8))

# Demand distribution (log-log scale)
plt.subplot(2, 1, 1)
plt.loglog(rank, demand, label="Demand per Token", color="blue")
plt.axvline(x=tail_threshold, color="red", linestyle="--", label="Tail Threshold")
plt.xlabel("Token Rank (Log Scale)")
plt.ylabel("Demand per Token (Log Scale)")
plt.title("Demand Distribution by Token Rank")
plt.legend()
plt.grid(True, which="both", linestyle="--", linewidth=0.5)

# Cumulative contribution
plt.subplot(2, 1, 2)
plt.plot(rank, cumulative_demand, label="Cumulative Contribution", color="green")
plt.axvline(x=tail_threshold, color="red", linestyle="--", label="Tail Threshold")
plt.axhline(y=tail_contribution, color="purple", linestyle="--", label=f"Tail Contribution ({tail_contribution:.1%})")
plt.xlabel("Token Rank")
plt.ylabel("Cumulative Contribution")
plt.title("Cumulative Contribution of Tokens")
plt.legend()
plt.grid(True, linestyle="--", linewidth=0.5)

plt.tight_layout()
plt.savefig("conceptual_model_token_visualization.png", dpi=300)
plt.show()

# Print tail contribution for reference
print(f"Contribution of the tail (bottom 90% tokens): {tail_contribution:.2%}")
```

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from collections import Counter

# Load data (assume train_df["text_prepro"] is already preprocessed)
# Tokenize the preprocessed text
tokens = [word for text in train_df["text_prepro"] for word in text.split()]

# Generate token frequencies
token_frequencies = Counter(tokens)
sorted_frequencies = np.array(sorted(token_frequencies.values(), reverse=True))



# Define Q-function for cumulative contributions
def Q_function(z, frequencies):
    """Compute the cumulative contributions up to threshold z."""
    return np.sum(frequencies[frequencies <= z])

# Compute Q(z) for the dataset
z_values = np.logspace(0, np.log10(max(sorted_frequencies)), num=50)
Q_values = [Q_function(z, sorted_frequencies) for z in z_values]

# Tail contributions (e.g., bottom 10% frequencies)
tail_threshold = np.percentile(sorted_frequencies, 10)  # Bottom 10%
Q_tail_values = [Q_function(z, sorted_frequencies[sorted_frequencies <= tail_threshold]) for z in z_values]

# Plot Q(z) and Q_tail(z)
plt.figure(figsize=(10, 6))
plt.plot(z_values, Q_values, label="Total Cumulative Contributions (Q(z))", color="black")
plt.plot(z_values, Q_tail_values, label="Tail Contributions (Q_tail(z))", linestyle="--", color="gray")
plt.xscale("log")
plt.xlabel("Threshold (z)")
plt.ylabel("Cumulative Contribution")
plt.title("Uniform Partitioning: Q(z) vs. Q_tail(z)")
plt.legend()
plt.grid(True, which="both", linestyle="--", linewidth=0.5)
plt.tight_layout()
plt.savefig("Uniform Partitioning.png")
plt.show()



import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from collections import Counter

# Load the dataset (ensure train_df is preloaded and contains "text_prepro")
# Simulated train_df["text_prepro"]
train_df = pd.DataFrame({"text_prepro": ["word1 word2 word3 word1 word2 word1", "word4 word5 word1 word6"]})

# Step 1: Extract token frequencies
tokens = []
train_df["text_prepro"].dropna().apply(lambda x: tokens.extend(x.split()))
token_counts = Counter(tokens)
sorted_frequencies = np.array(sorted(token_counts.values(), reverse=True))

# Step 2: Define the density function
def density_function(x, alpha=2.5):
    """Power-law density function."""
    return x ** -alpha if x > 0 else 0

# Step 3: Define G-function and Q-function
```

```python
def G_function(z, density_func):
    """G-function: Contribution of frequent events above z."""
    return sum(density_func(x) for x in range(int(z), len(sorted_frequencies) + 1))

def Q_function(z, density_func):
    """Q-function: Cumulative contribution of rare events below z."""
    return sum(x * density_func(x) for x in range(1, int(z) + 1))

# Step 4: Compute G(z) and Q(z) for the dataset
z_values = np.linspace(1, len(sorted_frequencies), 50)
G_values = [G_function(z, density_function) for z in z_values]
Q_values = [Q_function(z, density_function) for z in z_values]

# Step 5: Normalize G and Q for comparison
G_values_normalized = np.array(G_values) / max(G_values)
Q_values_normalized = np.array(Q_values) / max(Q_values)

# Step 6: Plot results
plt.figure(figsize=(10, 6))
plt.plot(z_values, G_values_normalized, label="Normalized G(z) (Frequent Events)", color="black", linestyle="--")
plt.plot(z_values, Q_values_normalized, label="Normalized Q(z) (Rare Events)", color="black", linestyle="-")
plt.xlabel("Threshold (z)", fontsize=12)
plt.ylabel("Normalized Contribution", fontsize=12)
plt.title("Lemma 1: Contribution Concentration in Long-Tail Markets", fontsize=14)
plt.grid(True, linestyle="--", linewidth=0.5)
plt.legend(fontsize=12)
plt.tight_layout()
plt.savefig("lemma1_concentration_long_tail.png", dpi=300)
plt.show()



# Simulate time-varying density
def density_over_time(x, t, alpha=1.5):
    """Simulate time-varying density with fluctuations."""
    return 1 / (x ** alpha) * (1 + 0.1 * np.sin(2 * np.pi * t / 10))

# Compute Q(z, t) over time
time_steps = np.arange(1, 11)  # 10 time steps
z_values = np.logspace(1, np.log10(len(sorted_frequencies)), 50)

Q_time_series = [
    [Q_function(z, lambda x: density_over_time(x, t)) for z in z_values]
    for t in time_steps
]

# Temporal analysis visualization
plt.figure(figsize=(10, 6))
for t, Q_values in enumerate(Q_time_series):
    plt.plot(z_values, Q_values, label=f"Time Step {t+1}", color="black", linestyle="--" if t % 2 else "-")
plt.xscale("log")
plt.xlabel("Threshold (z)")
plt.ylabel("Q(z)")
plt.title("Temporal Evolution of Q-Function")
plt.legend()
plt.grid(True, which="both", linestyle="--", linewidth=0.5)
plt.tight_layout()
plt.savefig("Temporal Evolution of Q-Function.png")
plt.show()



import numpy as np
import matplotlib.pyplot as plt

# Define density function (e.g., long-tail distribution)
```

```python
def density_function(x, alpha=2, k=1):
    return k * x ** -alpha if x >= 1 else 0

# Logarithmic Transformation for Q(z)
def Q_log_transformed(z, density_func):
    u_values = np.linspace(np.log(1), np.log(z), 1000)
    return np.trapz([density_func(np.exp(u)) * np.exp(u) for u in u_values], u_values)

# Normalized Transformation for G(z)
def G_normalized(z, density_func):
    v_values = np.linspace(0, 1, 1000)
    return z ** 2 * np.trapz([v * density_func(v * z) for v in v_values], v_values)

# Compute results
z_values = np.logspace(1, 3, 50)
Q_values_log = [Q_log_transformed(z, density_function) for z in z_values]
G_values_normalized = [G_normalized(z, density_function) for z in z_values]

# Plot results
plt.figure(figsize=(12, 6))
plt.plot(z_values, Q_values_log, label="Log-Transformed Q(z)", color="black",
linestyle="-")
plt.plot(z_values, G_values_normalized, label="Normalized G(z)", color="black",
linestyle="--")
plt.xscale("log")
plt.xlabel("Threshold (z)")
plt.ylabel("Cumulative Contribution")
plt.title("Variable Substitution in Q(z) and G(z)")
plt.legend()
plt.grid(True, which="both", linestyle="--", linewidth=0.5)
plt.tight_layout()
plt.savefig("substitution_analysis.png")
plt.show()



import numpy as np
import matplotlib.pyplot as plt

# Generate example frequency data
z_max = 1000  # Maximum threshold
n_partitions = 20  # Number of partitions
z_values = np.linspace(1, z_max, 1000)
frequencies = 1 / z_values  # Example long-tail distribution

# Define Q(z) and G(z)
def Q_function(z):
    return np.cumsum(frequencies[:z])

def G_function(z):
    return np.sum(frequencies[z:])

# Uniform partitioning
partitions = np.linspace(1, z_max, n_partitions + 1)
Q_contributions = []
G_contributions = []

for i in range(1, len(partitions)):
    z_start = int(partitions[i - 1])
    z_end = int(partitions[i])
    Q_contributions.append(Q_function(z_end)[-1] - Q_function(z_start)[-1])
    G_contributions.append(G_function(z_end))

# Plot Q(z) and G(z) contributions
plt.figure(figsize=(12, 6))
plt.bar(range(1, n_partitions + 1), Q_contributions, label="Q(z) Contributions",
alpha=0.7, color="black")
plt.bar(range(1, n_partitions + 1), G_contributions, label="G(z) Contributions",
alpha=0.7, hatch="//", color="gray")
```

```python
plt.xlabel("Partition Index")
plt.ylabel("Cumulative Contribution")
plt.title("Granular Contributions via Uniform Partitioning")
plt.legend()
plt.grid(True, linestyle="--", linewidth=0.5)
plt.tight_layout()
plt.savefig("granular_contributions_uniform_partition.png", dpi=300)
plt.show()



import numpy as np
import matplotlib.pyplot as plt

# Simulate relative frequencies over time
time_points = [1, 2, 3, 4, 5]  # Discrete time points
N = 10  # Number of subintervals
domain = np.linspace(0, 1, 1000)  # Domain

# Simulate density function f_t(t) at each time point
densities = {t: np.sin(2 * np.pi * domain * t) ** 2 + 0.1 for t in time_points}  #
Example density

# Compute v_i,n,t for each subinterval and time
subintervals = np.linspace(0, 1, N + 1)
frequencies_over_time = {t: [] for t in time_points}

for t in time_points:
    for i in range(len(subintervals) - 1):
        sub_start, sub_end = subintervals[i], subintervals[i + 1]
        freq = np.trapz(densities[t][(domain >= sub_start) & (domain < sub_end)],
                        domain[(domain >= sub_start) & (domain < sub_end)])
        frequencies_over_time[t].append(freq)

# Plot 1: Visualize v_i,n,t over time
plt.figure(figsize=(12, 6))
for t in time_points:
    plt.plot(range(1, N + 1), frequencies_over_time[t], label=f"Time {t}",
    linestyle="--", color="black")
plt.xlabel("Subinterval Index")
plt.ylabel("Relative Frequency (v_i,n,t)")
plt.title("Relative Frequencies Over Time (Emerging Trends)")
plt.legend()
plt.grid(color="gray", linestyle="--", linewidth=0.5)
plt.savefig("relative_frequencies_over_time.png", dpi=300, bbox_inches="tight")
plt.show()

# Identify Emerging Trends
trend_changes = {i: [] for i in range(1, N + 1)}
for i in range(N):
    for t in range(len(time_points) - 1):
        change = frequencies_over_time[time_points[t + 1]][i] -
        frequencies_over_time[time_points[t]][i]
        trend_changes[i + 1].append(change)

# Plot 2: Visualize Trend Changes
plt.figure(figsize=(12, 6))
for i, changes in trend_changes.items():
    plt.plot(time_points[1:], changes, label=f"Subinterval {i}", linestyle="-",
    color="black")
plt.xlabel("Time Points")
plt.ylabel("Change in v_i,n,t")
plt.title("Trend Changes in Relative Frequencies")
plt.legend()
plt.grid(color="gray", linestyle="--", linewidth=0.5)
plt.savefig("trend_changes_relative_frequencies.png", dpi=300, bbox_inches="tight")
plt.show()
```

```
import numpy as np
import matplotlib.pyplot as plt

# Define density function (e.g., long-tail distribution)
def density_function(x, alpha=2, k=1):
    return k * x ** -alpha if x >= 1 else 0

# Logarithmic Transformation for Q(z)
def Q_log_transformed(z, density_func):
    u_values = np.linspace(np.log(1), np.log(z), 1000)
    return np.trapz([density_func(np.exp(u)) * np.exp(u) for u in u_values], u_values)

# Normalized Transformation for G(z)
def G_normalized(z, density_func):
    v_values = np.linspace(0, 1, 1000)
    return z ** 2 * np.trapz([v * density_func(v * z) for v in v_values], v_values)

# Compute results
z_values = np.logspace(1, 3, 50)
Q_values_log = [Q_log_transformed(z, density_function) for z in z_values]
G_values_normalized = [G_normalized(z, density_function) for z in z_values]

# Plot results
plt.figure(figsize=(12, 6))
plt.plot(z_values, Q_values_log, label="Log-Transformed Q(z)", color="black",
linestyle="-")
plt.plot(z_values, G_values_normalized, label="Normalized G(z)", color="black",
linestyle="--")
plt.xscale("log")
plt.xlabel("Threshold (z)")
plt.ylabel("Cumulative Contribution")
plt.title("Variable Substitution in Q(z) and G(z)")
plt.legend()
plt.grid(True, which="both", linestyle="--", linewidth=0.5)
plt.tight_layout()
plt.savefig("substitution_analysis.png")
plt.show()
```

```
import numpy as np
import matplotlib.pyplot as plt

# Define density function (non-increasing)
def density_function(x, alpha=2, k=1):
    return k * x ** -alpha if x >= 1 else 0

# Variable transformations
def Q_log_transformed(z, density_func):
    u_values = np.linspace(np.log(1), np.log(z), 1000)
    return np.trapz([density_func(np.exp(u)) * np.exp(u) for u in u_values], u_values)

def G_normalized(z, density_func):
    v_values = np.linspace(0, 1, 1000)
    return z * np.trapz([density_func(v * z) * v for v in v_values], v_values)

# Compute and plot results
z_values = np.logspace(1, 3, 50)
Q_values_log = [Q_log_transformed(z, density_function) for z in z_values]
G_values_normalized = [G_normalized(z, density_function) for z in z_values]

plt.figure(figsize=(12, 6))
plt.plot(z_values, Q_values_log, label="Log-Transformed Q(z)", color="black",
linestyle="-")
plt.plot(z_values, G_values_normalized, label="Normalized G(z)", color="black",
linestyle="--")
plt.xscale("log")
plt.xlabel("Threshold (z)")
plt.ylabel("Cumulative Contribution")
```

```python
plt.title("Variable Change in Q(z) and G(z) for Improved Tail Analysis")
plt.legend()
plt.grid(True, which="both", linestyle="--", linewidth=0.5)
plt.tight_layout()
plt.savefig("variable_change_analysis.png")
plt.show()



# Define density function with inconsistency
def density_function_inconsistent(x):
    return np.random.uniform(0.9, 1.1) * density_function_non_increasing(x)

# Compute Q(z) and Q_tail(z) for inconsistent density
Q_values = [Q_function(z, lambda x: density_function_inconsistent(x)) for z in z_values]
Q_tail_values = [Q_tail_function(z, lambda x: density_function_inconsistent(x)) for z in z_values]

# Plot results
plt.figure(figsize=(12, 6))
plt.plot(z_values, Q_values, label="Total Contributions (Q(z))", color="black", linewidth=2)
plt.plot(z_values, Q_tail_values, label="Tail Contributions (Q_tail(z))", color="black", linestyle="--", linewidth=2)
plt.xscale("log")
plt.xlabel("Threshold (z)")
plt.ylabel("Cumulative Contribution")
plt.title("Inconsistent Density: Q(z) vs. Q_tail(z)")
plt.legend()
plt.grid(True, which="both", linestyle="--", linewidth=0.5)
plt.tight_layout()
plt.savefig("inconsistent_density_bw.png")
plt.show()



# Define density function for non-increasing behavior
def density_function_non_increasing(x, alpha=2, k=1):
    return k * x ** -alpha if x >= 1 else 0

# Compute Q(z) and Q_tail(z) for non-increasing density
Q_values = [Q_function(z, lambda x: density_function_non_increasing(x)) for z in z_values]
Q_tail_values = [Q_tail_function(z, lambda x: density_function_non_increasing(x)) for z in z_values]

# Plot results
plt.figure(figsize=(12, 6))
plt.plot(z_values, Q_values, label="Total Contributions (Q(z))", color="black", linewidth=2)
plt.plot(z_values, Q_tail_values, label="Tail Contributions (Q_tail(z))", color="black", linestyle="--", linewidth=2)
plt.xscale("log")
plt.xlabel("Threshold (z)")
plt.ylabel("Cumulative Contribution")
plt.title("Non-Increasing Density: Q(z) vs. Q_tail(z)")
plt.legend()
plt.grid(True, which="both", linestyle="--", linewidth=0.5)
plt.tight_layout()
plt.savefig("non_increasing_density_bw.png")
plt.show()



import numpy as np
import matplotlib.pyplot as plt

# Example dataset: token frequencies (replace with actual data)
tokens = [word for text in train_df["text_prepro"] for word in text.split()]
```

```python
token_frequencies = dict(Counter(tokens))

# Sort frequencies in descending order
sorted_frequencies = np.array(sorted(token_frequencies.values(), reverse=True))

# Define the density function
def density_function(x, sorted_frequencies):
    return 1 / len(sorted_frequencies) if x in range(1, len(sorted_frequencies)+1) else 0

# Compute Q(z) and Q_tail(z)
def Q_function(z, density_func):
    return sum(density_func(x, sorted_frequencies) for x in range(1, int(z)+1))

def Q_tail_function(z, density_func):
    return Q_function(len(sorted_frequencies), density_func) - Q_function(z,
    density_func)

# Compute values for Q(z) and Q_tail(z)
z_values = np.logspace(1, np.log10(len(sorted_frequencies)), 50)
Q_values = [Q_function(z, density_function) for z in z_values]
Q_tail_values = [Q_tail_function(z, density_function) for z in z_values]

# Plot results
plt.figure(figsize=(12, 6))
plt.plot(z_values, Q_values, label="Total Cumulative Contributions (Q(z))",
color="black", linewidth=2)
plt.plot(z_values, Q_tail_values, label="Tail Contributions (Q_tail(z))", color="black",
linestyle="--", linewidth=2)
plt.xscale("log")
plt.xlabel("Threshold (z)")
plt.ylabel("Cumulative Contribution")
plt.title("Uniform Partitioning: Q(z) vs. Q_tail(z)")
plt.legend()
plt.grid(True, which="both", linestyle="--", linewidth=0.5)
plt.tight_layout()
plt.savefig("uniform_partitioning_bw.png")
plt.show()



import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from collections import Counter
from scipy.stats import powerlaw
from scipy.optimize import curve_fit
from sklearn.metrics import mean_squared_error

# Tokenize the text and calculate frequencies
def tokenize_and_count(data):
    all_tokens = []
    data.dropna().apply(lambda text: all_tokens.extend(text.split()))
    return Counter(all_tokens)

# Plot Rank-Frequency on a Log-Log Scale
def plot_log_log_rank_frequency(freq_counts):
    sorted_counts = sorted(freq_counts.values(), reverse=True)
    ranks = np.arange(1, len(sorted_counts) + 1)
    plt.figure(figsize=(10, 6))
    plt.loglog(ranks, sorted_counts, marker="o", color = "black",linestyle="none",
    label="Observed Data")
    plt.xlabel("Rank (log scale)")
    plt.ylabel("Frequency (log scale)")
    plt.title("Log-Log Plot of Rank-Frequency Distribution")
    plt.grid(True, which="both", linestyle="--", linewidth=0.5)
    plt.legend()
    plt.savefig("long-tail.png")
    plt.show()

```

```
# Fit Power-Law Distribution
def fit_power_law(freq_counts):
    sorted_counts = np.array(sorted(freq_counts.values(), reverse=True))
    ranks = np.arange(1, len(sorted_counts) + 1)

    def power_law(x, alpha, beta):
        return beta * x ** -alpha

    params, _ = curve_fit(power_law, ranks, sorted_counts, maxfev=10000)
    fitted_alpha, fitted_beta = params

    # Compute the predicted values
    predicted = power_law(ranks, fitted_alpha, fitted_beta)

    # Plot the observed vs. fitted data
    plt.figure(figsize=(10, 6))
    plt.loglog(ranks, sorted_counts, marker="o", linestyle="none", color = "black",label="Observed Data")
    plt.loglog(ranks, predicted, label=f"Fitted Power-Law (α={fitted_alpha:.2f})", linestyle="--", color = "black")
    plt.xlabel("Rank (log scale)")
    plt.ylabel("Frequency (log scale)")
    plt.title("Power-Law Fit to Rank-Frequency Distribution")
    plt.grid(True, which="both", linestyle="--", linewidth=0.5)
    plt.savefig("power-law.png")
    plt.legend()
    plt.show()

    mse = mean_squared_error(sorted_counts, predicted)
    print(f"Fitted Power-Law Parameters: α = {fitted_alpha:.2f}, β = {fitted_beta:.2f}")
    print(f"Mean Squared Error of Fit: {mse:.2f}")

    return fitted_alpha, mse

# Calculate Gini Coefficient
def calculate_gini(freq_counts):
    frequencies = np.array(sorted(freq_counts.values()))
    n = len(frequencies)
    cumulative_sum = np.cumsum(frequencies)
    gini = (n + 1 - 2 * np.sum(cumulative_sum) / cumulative_sum[-1]) / n
    print(f"Gini Coefficient: {gini:.2f}")
    return gini

# Example Usage
# Assuming train_df["text_prepro"] contains the preprocessed text data
freq_counts = tokenize_and_count(train_df["text_prepro"])

# Visualize Rank-Frequency Distribution
plot_log_log_rank_frequency(freq_counts)

# Fit and Evaluate Power-Law Model
fit_alpha, mse = fit_power_law(freq_counts)

# Calculate Gini Coefficient
gini_coefficient = calculate_gini(freq_counts)

# Long-Tail Determination
if fit_alpha > 1 and gini_coefficient > 0.5:
    print("The data displays long-tail behavior.")
else:
    print("The data does not exhibit long-tail behavior.")



import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.feature_extraction.text import CountVectorizer
from nltk.corpus import stopwords
```

```
from nltk.stem import WordNetLemmatizer
from wordcloud import WordCloud

# Step 1: Preprocess Text Data
def preprocess_text(text):
    """Tokenize, remove stopwords, and lemmatize."""
    lemmatizer = WordNetLemmatizer()
    stop_words = set(stopwords.words("english"))
    tokens = text.lower().split()  # Tokenize and lowercase
    tokens = [word for word in tokens if word.isalpha() and word not in stop_words]
    tokens = [lemmatizer.lemmatize(word) for word in tokens]
    return " ".join(tokens)

train_df["processed_text"] = train_df["text_prepro"].apply(preprocess_text)

# Step 2: Ensure datetime column exists
train_df["at"] = pd.to_datetime(train_df["at"])

# Step 3: Group by time intervals (e.g., weekly)
train_df["week"] = train_df["at"].dt.to_period("W")
grouped_text = train_df.groupby("week")["processed_text"].apply(lambda x: " ".join(x))

# Step 4: Compute Term Frequencies
vectorizer = CountVectorizer()
term_matrix = vectorizer.fit_transform(grouped_text.values)
terms = vectorizer.get_feature_names_out()
term_frequencies = pd.DataFrame(term_matrix.toarray(), index=grouped_text.index, columns=terms)

# Step 5: Normalize Frequencies (Relative Frequencies)
normalized_frequencies = term_frequencies.div(term_frequencies.sum(axis=1), axis=0)

# Step 6: Identify Emerging Trends
# Calculate the average rate of frequency increase
frequency_trend = normalized_frequencies.diff().mean(axis=0).sort_values(ascending=False)
top_emerging_terms = frequency_trend.head(10).index

# Step 7: Visualize Emerging Trends - Time Series Plot
plt.figure(figsize=(12, 8))
for term in top_emerging_terms:
    plt.plot(normalized_frequencies.index.to_timestamp(), normalized_frequencies[term], label=term, color = "black")

plt.title("Emerging Trends Over Time")
plt.xlabel("Time (Weeks)")
plt.ylabel("Relative Frequency")
plt.legend(title="Terms")
plt.grid()
plt.savefig("emerging_trends.png")
plt.show()

# Step 8: Visualize Emerging Trends - Word Cloud
wordcloud = WordCloud(width=800, height=400, background_color="white").generate_from_frequencies(
    frequency_trend.to_dict()
)
plt.figure(figsize=(10, 6))
plt.imshow(wordcloud, interpolation="bilinear")
plt.axis("off")
plt.title("Emerging Terms Word Cloud")
plt.show()



import numpy as np
import pandas as pd
from collections import Counter
import matplotlib.pyplot as plt

```

```python
# Tokenize and count token frequencies
def get_token_frequencies(data):
    all_tokens = []
    for text in data:
        all_tokens.extend(text.split())  # Tokenize by splitting on spaces
    token_counts = Counter(all_tokens)
    return sorted(token_counts.items(), key=lambda x: x[1], reverse=True)  # Sort by frequency

# Compute partition-specific contributions
def partition_contributions(frequencies, num_partitions=10):
    total_tokens = sum(freq for token, freq in frequencies)  # Total frequency
    partition_size = len(frequencies) // num_partitions  # Number of tokens per partition

    partition_results = {
        "Partition": [],
        "Cumulative Contribution (Q)": [],
        "Weighted Contribution (G)": [],
    }

    for i in range(num_partitions):
        start_idx = i * partition_size
        end_idx = (i + 1) * partition_size if i != num_partitions - 1 else len(frequencies)

        partition = frequencies[start_idx:end_idx]
        partition_cumulative = sum(freq for token, freq in partition)
        partition_weighted = sum(rank * freq for rank, (token, freq) in enumerate(partition, start=start_idx + 1))

        partition_results["Partition"].append(f"Partition {i + 1}")
        partition_results["Cumulative Contribution (Q)"].append(partition_cumulative / total_tokens)
        partition_results["Weighted Contribution (G)"].append(partition_weighted / total_tokens)

    return pd.DataFrame(partition_results)

# Visualize the contributions
def visualize_partition_contributions(partition_df):
    x = np.arange(len(partition_df))

    plt.figure(figsize=(12, 6))
    plt.bar(x - 0.2, partition_df["Cumulative Contribution (Q)"], width=0.4, label="Cumulative Contribution (Q)")
    plt.bar(x + 0.2, partition_df["Weighted Contribution (G)"], width=0.4, label="Weighted Contribution (G)")
    plt.xticks(x, partition_df["Partition"], rotation=45)
    plt.xlabel("Partitions")
    plt.ylabel("Contribution")
    plt.title("Partition-Specific Contributions (Q and G)")
    plt.legend()
    plt.tight_layout()
    plt.show()

# Main Workflow
if __name__ == "__main__":
    # Load the data (replace with your dataset)
    token_frequencies = get_token_frequencies(train_df["text_prepro"])  # Extract token frequencies

    # Partition-specific contributions
    partition_df = partition_contributions(token_frequencies, num_partitions=10)

    # Visualize the results
    visualize_partition_contributions(partition_df)

    # Display the DataFrame
    print(partition_df)
```

```
import numpy as np
import matplotlib.pyplot as plt
from collections import Counter

# Tokenize and compute token frequencies
def compute_token_frequencies(data):
    all_tokens = " ".join(data).split()
    return Counter(all_tokens)

# Compute cumulative contribution Q(z)
def compute_Q_function(frequencies, thresholds):
    sorted_freqs = np.array(sorted(frequencies.values(), reverse=True))
    Q_values = [sum(sorted_freqs[:z]) for z in thresholds]
    return Q_values

# Detect new peaks in Q(z)
def detect_peaks(Q_values, thresholds, relative_increase=0.2):
    peaks = []
    for i in range(1, len(Q_values)):
        delta = Q_values[i] - Q_values[i-1]
        relative_change = delta / Q_values[i-1] if Q_values[i-1] > 0 else 0
        if relative_change > relative_increase:  # Significant jump
            peaks.append((thresholds[i], Q_values[i]))
    return peaks

# Main execution
data = train_df["text_prepro"]  # Replace with your column
token_frequencies = compute_token_frequencies(data)

# Define thresholds
thresholds = range(1, len(token_frequencies) + 1, 10)  # Every 10th token for efficiency

# Compute Q(z)
Q_values = compute_Q_function(token_frequencies, thresholds)

# Detect peaks
peaks = detect_peaks(Q_values, thresholds)

# Plot Q(z)
plt.figure(figsize=(10, 6))
plt.plot(thresholds, Q_values, '-o', label="Q(z)", markersize=4, color = 'black')
plt.xlabel("Threshold (z)")
plt.ylabel("Cumulative Contribution Q(z)")
plt.title("Q-Function with Detected Peaks")
plt.grid()
for z, Q in peaks:
    plt.axvline(x=z, color='black', linestyle='--', alpha=0.7)
    plt.text(z, Q, f"Peak @ z={z}", rotation=90, color='black')
plt.legend()
plt.savefig("Q-Function with Detected Peaks.png")
plt.show()

# Display peaks
print("Detected Peaks (Threshold z, Q(z)):")
for z, Q in peaks:
    print(f"Threshold z={z}, Q(z)={Q:.2f}")



from collections import Counter
import pandas as pd
import matplotlib.pyplot as plt

# Assuming token_frequencies is a Counter object (e.g., token_frequencies = 
Counter(tokens))
# Sort the tokens by frequency in descending order
```

```
sorted_tokens = pd.DataFrame(token_frequencies.items(), columns=["Token", "Frequency"])
sorted_tokens = sorted_tokens.sort_values(by="Frequency", ascending=False)

# Extract thresholds z=11 and z=21
threshold_z11 = sorted_tokens[:11]
threshold_z21 = sorted_tokens[:21]

# Display summaries
print("Summary of Tokens at z=11:")
print(threshold_z11)

print("\nSummary of Tokens at z=21:")
print(threshold_z21)

# Plot contributions at z=11
plt.figure(figsize=(10, 5))
plt.bar(threshold_z11["Token"], threshold_z11["Frequency"], color='black')
plt.title("Token Contributions at z=11")
plt.ylabel("Frequency")
plt.xlabel("Tokens")
plt.xticks(rotation=45)
plt.show()

# Plot contributions at z=21
plt.figure(figsize=(10, 5))
plt.bar(threshold_z21["Token"], threshold_z21["Frequency"], color='gray')
plt.title("Token Contributions at z=21")
plt.ylabel("Frequency")
plt.xlabel("Tokens")
plt.xticks(rotation=45)
plt.show()



from textblob import TextBlob
import pandas as pd
import matplotlib.pyplot as plt

# Check if train_df and the 'text_prepro' column exist
if "text_prepro" not in train_df.columns:
    raise ValueError("Column 'text_prepro' not found in train_df. Please check your dataset.")

# Ensure the column is not empty or full of null values
if train_df["text_prepro"].isnull().all():
    raise ValueError("Column 'text_prepro' is empty. Please ensure it contains preprocessed reviews.")

# Function to analyze sentiment for a single review
def analyze_sentiment(text):
    """
    Analyze sentiment of a given text using TextBlob.
    Returns 'Positive', 'Negative', or 'Neutral' based on polarity.
    """
    if not isinstance(text, str):
        return "Neutral"  # Default for non-text entries
    polarity = TextBlob(text).sentiment.polarity
    if polarity > 0:
        return "Positive"
    elif polarity < 0:
        return "Negative"
    else:
        return "Neutral"

# Apply sentiment analysis to the entire dataset
train_df["Sentiment"] = train_df["text_prepro"].apply(analyze_sentiment)

# Count the sentiment distribution
sentiment_distribution = train_df["Sentiment"].value_counts()
```

```
# Print sentiment distribution
print("Sentiment Distribution:\n", sentiment_distribution)

# Visualize sentiment distribution
plt.figure(figsize=(10, 6))
sentiment_distribution.plot(kind="bar", color=["black", "gray", "white"])
plt.title("Sentiment Distribution Across Reviews")
plt.xlabel("Sentiment")
plt.ylabel("Number of Reviews")
plt.xticks(rotation=0)
plt.grid(axis="y", linestyle="--", linewidth=0.5)
plt.tight_layout()
plt.savefig("sentiment_distribution_large_dataset.png", dpi=300)
plt.show()

# Display a few reviews for each sentiment category
for sentiment_class in ["Positive", "Negative", "Neutral"]:
    print(f"\nSample {sentiment_class} Reviews:")
    sample_reviews = train_df[train_df["Sentiment"] == sentiment_class]["text_prepro"].head(5)
    print(sample_reviews)



import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

# Simulated Data: Replace with actual results
data = {
    "Time Steps": [0, 1, 2, 3, 4, 5],
    "Q(z=10)": [10, 12, 15, 9, 6, 4],
    "Q(z=30)": [20, 25, 35, 50, 60, 75],
    "Q(z=50)": [30, 40, 60, 80, 100, 120],
}

# Convert data into a DataFrame
df = pd.DataFrame(data)

# Set Time Steps as the index for analysis
df.set_index("Time Steps", inplace=True)
print(df)

# Output:
#            Q(z=10)  Q(z=30)  Q(z=50)
# Time Steps
# 0               10       20       30
# 1               12       25       40
# 2               15       35       60
# 3                9       50       80
# 4                6       60      100
# 5                4       75      120



# Plotting Q(z) trends for each threshold
plt.figure(figsize=(10, 6))
for column in df.columns:
    plt.plot(df.index, df[column], marker='o', linestyle='-', label=column)

plt.title("Temporal Evolution of Q(z) Across Thresholds")
plt.xlabel("Time Steps")
plt.ylabel("Cumulative Contribution Q(z)")
plt.legend(title="Thresholds")
plt.grid()
plt.show()
```

```
# Compute the rate of change for Q(z)
rate_of_change = df.diff().dropna()

# Visualize rate of change
plt.figure(figsize=(10, 6))
for column in rate_of_change.columns:
    plt.plot(rate_of_change.index, rate_of_change[column], marker='o', color = 'black',
    linestyle='--', label=f"Rate of Change {column}")

plt.title("Rate of Change in Q(z) Over Time")
plt.xlabel("Time Steps")
plt.ylabel("Change in Q(z)")
plt.legend(title="Thresholds")
plt.grid()
plt.show()

print("Rate of Change Table:")
print(rate_of_change)



# Persistence Analysis: Summarize final cumulative contribution for each threshold
final_contribution = df.iloc[-1]

# Plot bar chart for persistence
plt.figure(figsize=(8, 5))
plt.bar(final_contribution.index, final_contribution.values, color='gray')
plt.title("Final Cumulative Contributions at Different Thresholds")
plt.xlabel("Thresholds (z)")
plt.ylabel("Q(z) at Final Time Step")
plt.savefig("Cumulative Contributions at Different Thresholds.png")
plt.show()

print("Final Contributions:")
print(final_contribution)



import plotly.express as px

# Melt the DataFrame for long-format plotting
long_df = df.reset_index().melt(id_vars="Time Steps", var_name="Threshold",
value_name="Q(z)")

# Create an interactive line plot
fig = px.line(long_df, x="Time Steps", y="Q(z)", color="Threshold", markers=True,
              title="Interactive Temporal Evolution of Q(z)",
              labels={"Q(z)": "Cumulative Contribution Q(z)"})

fig.show()



# Re-import necessary libraries after execution state reset
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from scipy.signal import find_peaks

# Simulated Q(z) data (replace with actual values or load precomputed data)
thresholds = np.arange(1, 101)
Q_z = np.cumsum(np.random.randint(1, 20, size=len(thresholds)))  # Simulated cumulative
Q(z)

# Find the peaks in Q(z) for optimal thresholds
peaks, _ = find_peaks(Q_z, prominence=10)
```

```python
# Plot Q(z) with detected peaks
plt.figure(figsize=(10, 6))
plt.plot(thresholds, Q_z, label="Cumulative Q(z)", color='black')
plt.scatter(thresholds[peaks], Q_z[peaks], color='red', zorder=5, label="Detected Peaks")
plt.title("Threshold Optimization for Q(z)")
plt.xlabel("Threshold (z)")
plt.ylabel("Cumulative Contribution Q(z)")
plt.legend()
plt.grid()
plt.show()

# Highlight thresholds of interest
optimal_thresholds = thresholds[peaks]
optimal_Q_values = Q_z[peaks]

threshold_opt_df = pd.DataFrame({
    "Threshold": optimal_thresholds,
    "Q(z)": optimal_Q_values
})

threshold_opt_df



from collections import Counter
from sklearn.preprocessing import StandardScaler

# Simulated token-frequency data for higher thresholds (replace with actual data)
token_frequencies = Counter({
    "rare_token_1": 50, "rare_token_2": 60, "common_token_1": 1000,
    "common_token_2": 950, "niche_token_1": 70, "niche_token_2": 80
})

# Convert token frequencies to DataFrame
token_df = pd.DataFrame(token_frequencies.items(), columns=["Token", "Frequency"])

# Preprocess data for clustering
X = token_df["Frequency"].values.reshape(-1, 1)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Apply K-Means clustering
kmeans = KMeans(n_clusters=2, random_state=42)
token_df["Cluster"] = kmeans.fit_predict(X_scaled)

# Visualize clusters
plt.figure(figsize=(10, 6))
for cluster in token_df["Cluster"].unique():
    cluster_data = token_df[token_df["Cluster"] == cluster]
    plt.bar(cluster_data["Token"], cluster_data["Frequency"], label=f"Cluster {cluster}")
plt.xticks(rotation=45)
plt.title("Cluster Analysis of Token Contributions")
plt.xlabel("Tokens")
plt.ylabel("Frequency")
plt.legend()
plt.show()

token_df
```