

```

1  import numpy as np
2  import pandas as pd
3  import tensorflow as tf
4  from tensorflow.keras.layers import Input, Embedding, Dense, MultiHeadAttention,
GlobalMaxPooling1D, LayerNormalization, Dropout
5  from tensorflow.keras.initializers import Constant
6  from tensorflow.keras.regularizers import l2
7  from tensorflow.keras import Model
8  from sklearn.model_selection import train_test_split
9  from sklearn.preprocessing import LabelEncoder
10 from keras.preprocessing.sequence import pad_sequences
11 from keras.callbacks import EarlyStopping
12 from keras.optimizers import Adam
13 from gensim.models import Word2Vec
14
15 # Load dataset
16 train_df = pd.read_csv("tinder.csv")
17
18 # Text Preprocessing Functions
19 import re
20 import string
21 import nltk
22 from nltk.corpus import stopwords
23 from nltk.tokenize import word_tokenize
24 from nltk.stem import WordNetLemmatizer
25 from bs4 import BeautifulSoup
26
27 nltk.download('stopwords')
28 nltk.download('punkt')
29 nltk.download('wordnet')
30
31 lemmatizer = WordNetLemmatizer()
32
33 def preprocess_text(text):
34     text = text.lower()
35     tokens = word_tokenize(text)
36     stopwords_set = set(stopwords.words('english'))
37     tokens = [lemmatizer.lemmatize(word) for word in tokens if word not in stopwords_set]
38     return ' '.join(tokens)
39
40 def preprocess_text2(text):
41     text = text.lower()
42     tokens = word_tokenize(text)
43     return tokens
44
45 def clean_text(text):
46     text = BeautifulSoup(text, "html.parser").get_text()
47     text = re.sub(r'http\S+', '', text)
48     text = re.sub(r'^\x00-\x7F.', '', text)
49     text = re.sub(f'[{re.escape(string.punctuation.replace(".", ""))}]', '', text)
50     text = re.sub(r'\b\d+\b', '', text)
51     text = re.sub(r'\.{2,}', '', text)
52     text = re.sub(r'(?<=\.)\s+', '', text).strip()
53     return text
54
55 def remove_repeated_text(text):
56     pattern = r'\b(\w+\s?) (\.\s?\1){2,}\b'
57     return re.sub(pattern, '', text)
58
59 def remove_repeated_text2(text):
60     pattern = re.compile(r'\b(\w+)\b\s+\1(?:\s+\1)+\b', re.IGNORECASE)
61     def remove_repeats(match):
62         return match.group(1)
63     return pattern.sub(remove_repeats, text)
64
65 def remove_repeating_pattern(text):
66     pattern = r'(\w)\1+'
67     return re.sub(pattern, '', text)
68

```

```

69 # Ensure all values in 'content' are strings and handle missing values
70 train_df["content"] = train_df["content"].fillna("").astype(str)
71
72 # Apply preprocessing
73 train_df["remove_repeat_word"] =
74 train_df["content"].apply(remove_repeated_text).apply(remove_repeated_text2)
75 train_df["clean_text"] =
76 train_df["remove_repeat_word"].apply(clean_text).apply(remove_repeating_pattern)
77 train_df["text_prepro"] = train_df["clean_text"].apply(preprocess_text)
78
79 import numpy as np
80 import pandas as pd
81 import matplotlib.pyplot as plt
82
83 # Vocabulary calculation functions
84 def calculate_vocabulary(tokens, N):
85     """Calculate vocabulary size for the first N tokens."""
86     return len(set(tokens[:N]))
87
88 # Vocabulary growth models
89 def heaps_law(N, k=20, beta=0.6):
90     """Vocabulary size based on Heap's law (LNRE)."""
91     return k * (N ** beta)
92
93 def lstm_vocab_growth(N):
94     """
95     Simulate LSTM vocabulary growth.
96     This assumes diminishing returns due to training bias toward frequent words.
97     """
98     return np.log(N) ** 2
99
100 def bert_vocab_growth(N, max_vocab=30000):
101     """
102     Simulate BERT vocabulary growth.
103     BERT relies on subword tokenization, so the vocabulary growth saturates early.
104     """
105     return max_vocab * (1 - np.exp(-N / max_vocab))
106
107 def laplace_vocab_growth(N, alpha=1, total_vocab=5000):
108     """
109     Simulate vocabulary growth using Laplace smoothing.
110     """
111     return total_vocab * (1 - np.exp(-N / (alpha * total_vocab)))
112
113 def katz_vocab_growth(N, d=0.5, total_vocab=5000):
114     """
115     Simulate vocabulary growth using Katz Backoff.
116     """
117     return total_vocab * (1 - d * np.exp(-N / (total_vocab)))
118
119 train_df = pd.DataFrame({"text_prepro": text_prepro_list})
120
121 # Tokenize the text data
122 tokens = []
123 train_df["text_prepro"].dropna().apply(lambda x: tokens.extend(x.split()))
124
125 # Token counts for analysis
126 N_values = np.logspace(3, 6, num=50, dtype=int) # Token counts from 1,000 to 1,000,000
127
128 # Calculate actual vocabulary sizes
129 actual_vocab_sizes = [calculate_vocabulary(tokens, N) for N in N_values]
130
131 # Simulate LNRE (Heap's Law), LSTM, and BERT growth for comparison
132 lnre_vocab = [heaps_law(N) for N in N_values]
133 lstm_vocab = [lstm_vocab_growth(N) for N in N_values]
134 bert_vocab = [bert_vocab_growth(N) for N in N_values]
135
136 # Plotting the results

```

```

136 plt.figure(figsize=(12, 6))
137 plt.plot(N_values, lnre_vocab, label="LNRE (Heap's Law)", color="black",
138         linestyle="solid", linewidth=2)
139 plt.plot(N_values, lstm_vocab, label="LSTM", color="black", linestyle="dotted",
140         linewidth=2)
141 plt.plot(N_values, bert_vocab, label="BERT", color="black", linestyle="dashed",
142         linewidth=2)
143 plt.plot(N_values, actual_vocab_sizes, label="Dataset (Actual)", linestyle="dashdot",
144         color="black", linewidth=2)
145
146 # Log scale for better visualization
147 plt.xscale("log")
148 plt.yscale("log")
149
150 # Adding labels and legend
151 plt.xlabel("Number of Tokens (N)", fontsize=12)
152 plt.ylabel("Vocabulary Size (V)", fontsize=12)
153 plt.title("Vocabulary Growth Comparison: LNRE vs LSTM vs BERT", fontsize=14)
154 plt.legend(fontsize=12)
155 plt.grid(True, which="both", linestyle="--", linewidth=0.5)
156 plt.tight_layout()
157 plt.savefig("vocabulary_growth_comparison_test_1.png", dpi=300)
158 plt.show()
159
160
161 import numpy as np
162 import matplotlib.pyplot as plt
163 from collections import Counter
164
165 # Example: Simulate tokenized data (replace with your actual tokenized dataset)
166 tokens = ["word1", "word2", "word3", "word1", "word2", "word4", "word1", "word5"]
167 token_counts = Counter(tokens)
168
169 # Sort tokens by frequency
170 sorted_token_counts = sorted(token_counts.values(), reverse=True)
171
172 # Plot the frequency distribution
173 plt.figure(figsize=(10, 6))
174 plt.plot(sorted_token_counts, label="Token Frequency Distribution", color = "black")
175 plt.xlabel("Token Rank")
176 plt.ylabel("Frequency")
177 plt.title("Token Frequency Distribution")
178 plt.grid(True)
179 plt.tight_layout()
180 plt.savefig("tail_behav.png", dpi=300)
181 plt.show()
182
183 # Calculate token ranks
184 ranks = np.arange(1, len(sorted_token_counts) + 1)
185
186 # Plot on a log-log scale
187 plt.figure(figsize=(10, 6))
188 plt.loglog(ranks, sorted_token_counts, marker="o", color = "black",
189         label="Rank-Frequency Plot")
190 plt.xlabel("Rank (Log Scale)")
191 plt.ylabel("Frequency (Log Scale)")
192 plt.title("Log-Log Plot of Token Rank vs. Frequency")
193 plt.grid(True, which="both", linestyle="--", linewidth=0.5)
194 plt.tight_layout()
195 plt.savefig("tail_behav_2.png", dpi=300)
196 plt.show()
197 plt.legend()
198 plt.show()
199
200 import numpy as np
201 import matplotlib.pyplot as plt
202 from collections import Counter
203
204 # Define vocabulary growth models

```

```

200 def heaps_law(N, k=20, beta=0.6):
201     return k * (N ** beta)
202
203 def lstm_vocab_growth(N):
204     return np.log(N) ** 2
205
206 def bert_vocab_growth(N, max_vocab=30000):
207     return max_vocab * (1 - np.exp(-N / max_vocab))
208
209 def laplace_vocab_growth(N, alpha=1, total_vocab=5000):
210     return total_vocab * (1 - np.exp(-N / (alpha * total_vocab)))
211
212 def katz_vocab_growth(N, d=0.5, total_vocab=5000):
213     return total_vocab * (1 - d * np.exp(-N / (total_vocab)))
214
215 def calculate_vocabulary(tokens, N):
216     return len(set(tokens[:N]))
217
218 # Simulated token counts
219 N_values = np.logspace(3, 6, num=50, dtype=int)
220
221 # Ensure tokens are defined (replace with your actual data extraction)
222 tokens = []
223 train_df["text_prepro"].dropna().apply(lambda x: tokens.extend(x.split()))
224
225 # Plotting all vocabulary growth models
226 plt.figure(figsize=(12, 6))
227
228 # LNRE (Heap's Law)
229 lnre_vocab = [heaps_law(N) for N in N_values]
230 plt.plot(N_values, lnre_vocab, label="LNRE (Heap's Law)", color="black", linestyle =
"solid", linewidth=2)
231
232 # LSTM Vocabulary Growth
233 lstm_vocab = [lstm_vocab_growth(N) for N in N_values]
234 plt.plot(N_values, lstm_vocab, label="LSTM", color="black", linestyle="dashed",
linewidth=2)
235
236 # BERT Vocabulary Growth
237 bert_vocab = [bert_vocab_growth(N) for N in N_values]
238 plt.plot(N_values, bert_vocab, label="BERT", color="black", linestyle="--", linewidth=2)
239
240 # Laplace Smoothing Vocabulary Growth
241 laplace_vocab = [laplace_vocab_growth(N) for N in N_values]
242 plt.plot(N_values, laplace_vocab, label="Laplace Smoothing", color="black",
linestyle=":", linewidth=2)
243
244 # Katz Backoff Vocabulary Growth
245 katz_vocab = [katz_vocab_growth(N) for N in N_values]
246 plt.plot(N_values, katz_vocab, label="Katz Backoff", color="black", linestyle="-. ",
linewidth=2)
247
248 # Adding actual vocabulary sizes
249 actual_vocab_sizes = [calculate_vocabulary(tokens, N) for N in N_values]
250 plt.plot(N_values, actual_vocab_sizes, label="Dataset (Actual)", color="black",
linestyle="--", linewidth=2)
251
252 # Log scale for better visualization
253 plt.xscale("log")
254 plt.yscale("log")
255
256 # Adding labels and legend
257 plt.xlabel("Number of Tokens (N)", fontsize=12)
258 plt.ylabel("Vocabulary Size (V)", fontsize=12)
259 plt.title("Vocabulary Growth Comparison: All Models", fontsize=14)
260 plt.legend(fontsize=12)
261 plt.grid(True, which="both", linestyle="--", linewidth=0.5)
262 plt.tight_layout()
263 plt.savefig("vocabulary_growth_comparison_all_models.png", dpi=300)

```

```

264 plt.show()
265
266 import numpy as np
267 import matplotlib.pyplot as plt
268
269 # Define vocabulary growth models
270 def heaps_law(N, k=20, beta=0.6):
271     return k * (N ** beta)
272
273 def lstm_vocab_growth(N):
274     return np.log(N) ** 2
275
276 def bert_vocab_growth(N, max_vocab=30000):
277     return max_vocab * (1 - np.exp(-N / max_vocab))
278
279 def laplace_vocab_growth(N, alpha=1, total_vocab=5000):
280     return total_vocab * (1 - np.exp(-N / (alpha * total_vocab)))
281
282 def katz_vocab_growth(N, d=0.5, total_vocab=5000):
283     return total_vocab * (1 - d * np.exp(-N / (total_vocab)))
284
285 def calculate_vocabulary(tokens, N):
286     return len(set(tokens[:N]))
287
288 # Simulated token counts
289 N_values = np.logspace(3, 6, num=50, dtype=int)
290
291 # Plotting all vocabulary growth models
292 plt.figure(figsize=(12, 6))
293
294 # LNRE (Heap's Law)
295 lnre_vocab = [heaps_law(N) for N in N_values]
296 plt.plot(N_values, lnre_vocab, label="LNRE (Heap's Law)", color='black',
297         linestyle='solid', linewidth=2)
298
299 # LSTM Vocabulary Growth
300 lstm_vocab = [lstm_vocab_growth(N) for N in N_values]
301 plt.plot(N_values, lstm_vocab, label="LSTM", color='black', linestyle='dashed',
302         linewidth=2)
303
304 # BERT Vocabulary Growth
305 bert_vocab = [bert_vocab_growth(N) for N in N_values]
306 plt.plot(N_values, bert_vocab, label="BERT", color='black', linestyle='dotted',
307         linewidth=2)
308
309 # Laplace Smoothing Vocabulary Growth
310 laplace_vocab = [laplace_vocab_growth(N) for N in N_values]
311 plt.plot(N_values, laplace_vocab, label="Laplace Smoothing", color='black',
312         linestyle='dashdot', linewidth=2)
313
314 # Katz Backoff Vocabulary Growth
315 katz_vocab = [katz_vocab_growth(N) for N in N_values]
316 plt.plot(N_values, katz_vocab, label="Katz Backoff", color='black', linestyle=(0, (5,
317         10)), linewidth=2)
318
319 # Adding actual vocabulary sizes
320 actual_vocab_sizes = [calculate_vocabulary([], N) for N in N_values] # Replace with
321 actual token data
322 plt.plot(N_values, actual_vocab_sizes, label="Dataset (Actual)", color='black',
323         linestyle=(0, (3, 5, 1, 5)), linewidth=2)
324
325 # Log scale for better visualization
326 plt.xscale("log")
327 plt.yscale("log")
328
329 # Adding labels and legend
330 plt.xlabel("Number of Tokens (N)", fontsize=12)
331 plt.ylabel("Vocabulary Size (V)", fontsize=12)
332 plt.title("Vocabulary Growth Comparison: All Models", fontsize=14)

```

```

326 plt.legend(fontsize=10, loc='upper left')
327 plt.grid(True, which="both", linestyle="--", linewidth=0.5)
328 plt.tight_layout()
329
330 # Save and show the plot
331 plt.savefig("vocabulary_growth_comparison_bw_final.png", dpi=300)
332 plt.show()
333
334 from collections import Counter
335
336 # Generate token frequencies from the dataset
337 # Ensure train_df["text_prepro"] contains tokenized or cleaned text
338 token_frequencies = Counter()
339
340 # Split preprocessed text into tokens and count frequencies
341 train_df["text_prepro"].dropna().apply(lambda x: token_frequencies.update(x.split()))
342
343 # Now you can use token_frequencies in your function
344 rare_words = get_rare_words(token_frequencies, threshold=5)
345 print("Rare Words:", rare_words)
346
347
348 import numpy as np
349 from scipy.integrate import quad
350
351 # Define the G-function (e.g., from token frequencies)
352 def G_function(z, token_frequencies):
353     """G-function for a given z and token frequencies."""
354     return sum(freq for token, freq in token_frequencies.items() if freq > z)
355
356 # Define the Q-function
357 def Q_function(z, token_frequencies):
358     """Q-function as the integral of x * G_function(x)."""
359     def integrand(x):
360         return x * G_function(x, token_frequencies)
361
362     Q_value, _ = quad(integrand, 0, z) # Integrate from 0 to z
363     return Q_value
364
365 # Function to extract rare words
366 def get_rare_words(token_frequencies, threshold=5):
367     """Identify rare words based on a frequency threshold."""
368     return {token: freq for token, freq in token_frequencies.items() if freq <=
369             threshold}
370
371 # Example use case
372 rare_words = get_rare_words(token_frequencies, threshold=5)
373 print("Rare Words:", rare_words)
374
375 # Simulate Q-function over time (e.g., for different time windows)
376 def analyze_trends_over_time(token_frequencies_time_series, z_values):
377     """Analyze changes in Q-function over time."""
378     Q_values_over_time = []
379
380     for time_step, freq_distribution in enumerate(token_frequencies_time_series):
381         Q_values = [Q_function(z, freq_distribution) for z in z_values]
382         Q_values_over_time.append(Q_values)
383
384     return Q_values_over_time
385
386 import matplotlib.pyplot as plt
387
388 def visualize_Q_function(z_values, Q_values, title="Q-Function Visualization"):
389     """Plot Q-function values."""
390     plt.figure(figsize=(10, 6))
391     plt.plot(z_values, Q_values, marker="o", label="Q-function", color = "black")
392     plt.xlabel("z")
393     plt.ylabel("Q(z)")
394     plt.title(title)

```

```

394     plt.grid(True)
395     plt.legend()
396     plt.show()
397
398 # Example token frequencies for one dataset
399 token_frequencies = {
400     "word1": 100, "word2": 50, "word3": 5, "word4": 1, "word5": 2
401 }
402
403 # Calculate Q-function for a range of z-values
404 z_values = np.linspace(1, 50, 10)
405 Q_values = [Q_function(z, token_frequencies) for z in z_values]
406
407 # Visualize Q-function
408 visualize_Q_function(z_values, Q_values, title="Q-Function Example")
409
410 # Track rare words
411 rare_words = get_rare_words(token_frequencies, threshold=5)
412 print("Rare Words:", rare_words)
413
414 # Example: Analyze trends with synthetic time series data
415 time_series = [
416     {"word1": 100, "word2": 50, "word3": 5, "word4": 1}, # Time step 1
417     {"word1": 90, "word2": 40, "word3": 6, "word4": 2}, # Time step 2
418     {"word1": 80, "word2": 30, "word3": 8, "word4": 3}   # Time step 3
419 ]
420 Q_trends = analyze_trends_over_time(time_series, z_values)
421
422 # Visualize trends
423 for i, Q in enumerate(Q_trends):
424     visualize_Q_function(z_values, Q, title=f"Q-Function at Time Step {i + 1}")
425
426
427 import numpy as np
428 import matplotlib.pyplot as plt
429
430 # Define G-function
431 def G_function(z, token_frequencies):
432     """G-function as the sum of probabilities greater than z."""
433     return sum(freq for token, freq in token_frequencies.items() if freq > z)
434
435 # Define Q-function
436 def Q_function(z, token_frequencies):
437     """Q-function as the cumulative contribution of probabilities below z."""
438     return sum(freq for token, freq in token_frequencies.items() if freq <= z)
439
440
441 # Generate z-values (thresholds)
442 z_values = np.linspace(1, max(token_frequencies.values()), 100)
443
444 # Compute Q-function for each z
445 Q_values = [Q_function(z, token_frequencies) for z in z_values]
446
447 # Plot the Q-function
448 plt.figure(figsize=(10, 6))
449 plt.plot(z_values, Q_values, marker="o", color = "black", label="Q-function")
450 plt.xlabel("Threshold (z)")
451 plt.ylabel("Q(z)")
452 plt.title("Q-Function Trends Across Thresholds")
453 plt.grid(True)
454 plt.legend()
455 plt.savefig("Q-Function Trends Across Thresholds.png")
456 plt.show()
457
458 time_series = [
459     {"word1": 100, "word2": 50, "word3": 5}, # Time step 1
460     {"word1": 90, "word2": 40, "word3": 8}, # Time step 2
461     {"word1": 80, "word2": 30, "word3": 12}  # Time step 3
462 ]

```

```

463
464 # Track Q-function over time
465 z = 10 # Fixed threshold
466 Q_trends = [Q_function(z, freq_distribution) for freq_distribution in time_series]
467
468 # Plot the Q-function trend over time
469 plt.figure(figsize=(10, 6))
470 plt.plot(range(len(Q_trends)), Q_trends, marker="o", color = "black",label=f"Q(z={z})
over time")
471 plt.xlabel("Time Steps")
472 plt.ylabel("Q(z)")
473 plt.title("Q-Function Trends Over Time")
474 plt.grid(True)
475 plt.legend()
476 plt.savefig("Q-Function Trends Across Time.png")
477 plt.show()
478
479
480 import matplotlib.pyplot as plt
481
482 # Example Q_function for demonstration (replace with your actual implementation)
483 def Q_function(z, freq_distribution):
484     return sum(f**z for f in freq_distribution) / len(freq_distribution)
485
486 # Example time series data (replace with your actual data)
487 time_series = [
488     [1, 2, 3], # Frequency distribution at time step 1
489     [2, 3, 4], # Frequency distribution at time step 2
490     [3, 4, 5], # And so on
491     [4, 5, 6],
492     [5, 6, 7],
493 ]
494
495 # Plot Q-functions for multiple z-values with distinct line styles
496 plt.figure(figsize=(10, 6))
497 line_styles = ['solid', 'dashed', 'dotted'] # Line styles for each z-value
498 markers = ['o', 's', 'd'] # Markers for each z-value
499
500 for i, z in enumerate([5, 10, 20]):
501     Q_values = [Q_function(z, freq_distribution) for freq_distribution in time_series]
502     plt.plot(
503         range(len(Q_values)),
504         Q_values,
505         linestyle=line_styles[i],
506         marker=markers[i],
507         label=f"Q(z={z})",
508         color='black', # Ensure black-and-white compliance
509         linewidth=2
510     )
511
512 # Add labels, grid, and legend
513 plt.xlabel("Time Steps", fontsize=12)
514 plt.ylabel("Q(z)", fontsize=12)
515 plt.title("Q-Function Trends for Multiple Thresholds", fontsize=14)
516 plt.grid(True, linestyle="--", linewidth=0.5)
517 plt.legend(fontsize=10, loc='upper left')
518
519 # Save and show the plot
520 plt.tight_layout()
521 plt.savefig("Q-Function_Trends_Across_Multiple_Thresholds_BW.png", dpi=300)
522 plt.show()
523
524 import matplotlib.pyplot as plt
525
526 # Example Q_function for demonstration (replace with your actual implementation)
527 def Q_function(z, freq_distribution):
528     return sum(f**z for f in freq_distribution) / len(freq_distribution)
529
530 # Example time series data (replace with your actual data)

```



```

531 time_series = [
532     [1, 2, 3], # Frequency distribution at time step 1
533     [2, 3, 4], # Frequency distribution at time step 2
534     [3, 4, 5], # And so on
535     [4, 5, 6],
536     [5, 6, 7],
537 ]
538
539 # Plot Q-functions for multiple z-values with distinct line styles
540 plt.figure(figsize=(10, 6))
541 line_styles = ['solid', 'dashed', 'dotted'] # Line styles for each z-value
542 markers = ['o', 's', 'd'] # Markers for each z-value
543
544 for i, z in enumerate([5, 10, 20]):
545     Q_values = [Q_function(z, freq_distribution) for freq_distribution in time_series]
546     plt.plot(
547         range(len(Q_values)),
548         Q_values,
549         linestyle=line_styles[i],
550         marker=markers[i],
551         label=f"Q(z={z})",
552         color='black', # Ensure black-and-white compliance
553         linewidth=2
554     )
555
556 # Add labels, grid, and legend
557 plt.xlabel("Time Steps", fontsize=12)
558 plt.ylabel("Q(z)", fontsize=12)
559 plt.title("Q-Function Trends for Multiple Thresholds", fontsize=14)
560 plt.grid(True, linestyle="--", linewidth=0.5)
561 plt.legend(fontsize=10, loc='upper left')
562
563 # Save and show the plot
564 plt.tight_layout()
565 plt.savefig("Q-Function_Trends_Across_Multiple_Thresholds_BW.png", dpi=300)
566 plt.show()
567
568
569 from mpl_toolkits.mplot3d import Axes3D
570 import numpy as np
571 import matplotlib.pyplot as plt
572
573 # Define Q-function
574 def Q_function(z, token_frequencies):
575     """Q-function as the cumulative contribution of probabilities below z."""
576     return sum(freq for token, freq in token_frequencies.items() if freq <= z)
577
578 # Simulated time-series data (replace with real data)
579 time_series = [
580     {"word1": 100, "word2": 50, "word3": 5, "word4": 1},
581     {"word1": 90, "word2": 40, "word3": 8, "word4": 2},
582     {"word1": 80, "word2": 30, "word3": 12, "word4": 4},
583 ]
584
585 # Generate z-values and compute Q-function over time
586 z_values = np.linspace(1, 100, 50)
587 Q_values_over_time = [
588     [Q_function(z, freq_distribution) for z in z_values] for freq_distribution in
589     time_series
590 ]
591
592 # Create 3D plot
593 fig = plt.figure(figsize=(10, 6))
594 ax = fig.add_subplot(111, projection='3d')
595
596 time_steps = np.arange(len(time_series))
597 Z, T = np.meshgrid(z_values, time_steps)
598 Q = np.array(Q_values_over_time)
599

```

```

599 # Use wireframe instead of a colored surface for black-and-white compatibility
600 ax.plot_wireframe(Z, T, Q, color='black', linewidth=0.8)
601
602 # Add labels and title
603 ax.set_xlabel("Threshold (z)", fontsize=10)
604 ax.set_ylabel("Time Steps", fontsize=10)
605 ax.set_zlabel("Q(z)", fontsize=10)
606 ax.set_title("3D Q-Function Trends Over Time", fontsize=12)
607
608 # Save and display the plot
609 plt.tight_layout()
610 plt.savefig("3D_Q_Function_Trends_BW.png", dpi=300, bbox_inches='tight')
611 plt.show()
612
613
614
615 import plotly.graph_objects as go
616 import numpy as np
617
618 # Define Q-function
619 def Q_function(z, token_frequencies):
620     """Q-function as the cumulative contribution of probabilities below z."""
621     return sum(freq for token, freq in token_frequencies.items() if freq <= z)
622
623 # Simulated time-series data (replace with real data)
624 time_series = [
625     {"word1": 100, "word2": 50, "word3": 5, "word4": 1},
626     {"word1": 90, "word2": 40, "word3": 8, "word4": 2},
627     {"word1": 80, "word2": 30, "word3": 12, "word4": 4},
628 ]
629
630 # Generate Q-function values
631 z_values = np.linspace(1, 100, 50)
632 time_steps = np.arange(len(time_series))
633 Q_values_over_time = [
634     [Q_function(z, freq_distribution) for z in z_values] for freq_distribution in
        time_series
635 ]
636
637 # Create a static plot with distinct dash patterns
638 fig = go.Figure()
639
640 dash_styles = ["solid", "dash", "dot"] # Different line styles for time steps
641
642 for t, (Q_vals, dash_style) in enumerate(zip(Q_values_over_time, dash_styles)):
643     fig.add_trace(go.Scatter(
644         x=z_values, y=Q_vals,
645         mode='lines',
646         line=dict(dash=dash_style, color='black', width=2),
647         name=f"Time Step {t+1}"
648     ))
649
650 fig.update_layout(
651     title="Static Q-Function Visualization for Different Time Steps",
652     xaxis_title="Threshold (z)",
653     yaxis_title="Q(z)",
654     legend_title="Time Step",
655     template="plotly_white",
656     font=dict(size=12),
657 )
658
659 # Save the figure as an image (optional)
660 fig.write_image("Static_Q_Function_Trends_BW.png", width=800, height=600)
661
662 # Show the plot
663 fig.show()
664
665
666

```

```

667 from matplotlib.animation import FuncAnimation
668
669 # Create figure
670 fig, ax = plt.subplots(figsize=(10, 6))
671 line, = ax.plot([], [], label="Q-function", color = "black")
672 ax.set_xlim(1, max(token_frequencies.values()))
673 ax.set_ylim(0, max(Q_values_over_time[-1]))
674 ax.set_xlabel("Threshold (z)")
675 ax.set_ylabel("Q(z)")
676 ax.set_title("Animated Q-Function Trends")
677 ax.grid(True)
678 ax.legend()
679
680 # Update function for animation
681 def animate(frame):
682     Q_vals = Q_values_over_time[frame]
683     line.set_data(z_values, Q_vals)
684     ax.set_title(f"Time Step {frame + 1}")
685     return line,
686
687 # Create animation
688 ani = FuncAnimation(fig, animate, frames=len(time_series), interval=1000, blit=True)
689 plt.show()
690
691
692 import numpy as np
693 import matplotlib.pyplot as plt
694
695 # Sample Q-function values for different thresholds z
696 thresholds = np.linspace(1, 100, 100)
697 Q_values = [np.log10(z) * z if z < 50 else z ** 0.8 for z in thresholds]
698
699 # Log-Log Plot
700 plt.figure(figsize=(10, 6))
701 plt.plot(thresholds, Q_values, label="Q-function", marker='o', linestyle='-',
702         color='black')
703 plt.xscale("log")
704 plt.yscale("log")
705 plt.xlabel("Threshold (z) (Log Scale)", fontsize=12)
706 plt.ylabel("Q(z) (Log Scale)", fontsize=12)
707 plt.title("Log-Log Plot of Q-Function", fontsize=14)
708 plt.grid(True, which="both", linestyle="--", linewidth=0.5)
709 plt.legend()
710 plt.tight_layout()
711 plt.savefig("Log-Log Plot of Q-function.png")
712 plt.show()
713
714 import numpy as np
715
716 # Step 1: Compute token frequencies
717 token_frequencies = {"word1": 3, "word2": 7, "word3": 1, "word4": 5, "word5": 2}
718
719 # Step 2: Define thresholds
720 thresholds = range(1, 10)
721
722 # Step 3: Compute Q-function values
723 def compute_q_function(token_frequencies, thresholds):
724     q_values = []
725     for z in thresholds:
726         q_value = sum(freq for freq in token_frequencies.values() if freq <= z)
727         q_values.append(q_value)
728     return q_values
729
730 q_values = compute_q_function(token_frequencies, thresholds)
731
732 # Step 4: Identify rare events
733 def identify_rare_events(data, q_function_values, threshold=5):
734     rare_indices = [i for i, q_val in enumerate(q_function_values) if q_val < threshold]
735     rare_events = [data[i] for i in rare_indices]

```

```

735     return rare_events
736
737 rare_events = identify_rare_events(list(token_frequencies.keys()), q_values, threshold=5)
738
739 print("Rare Events:", rare_events)
740
741
742 import numpy as np
743 import matplotlib.pyplot as plt
744
745 # Simulated time-series Q-function values for different thresholds
746 time_steps = np.arange(0, 3, 0.5) # Example time steps
747 thresholds = [10, 30, 50] # Specific thresholds to analyze
748 Q_time_series = {
749     10: [10, 12, 15, 10, 5, 0], # Q(z=10) over time
750     30: [20, 25, 30, 40, 50, 55], # Q(z=30) over time
751     50: [30, 35, 40, 60, 80, 100] # Q(z=50) over time
752 }
753
754 # Define line styles for each threshold
755 line_styles = ['solid', 'dotted', 'dashdot'] # Different line styles for thresholds
756 markers = ['o', 's', 'd'] # Different markers for thresholds
757
758 # Plot Temporal Trends
759 plt.figure(figsize=(10, 6))
760
761 for (z, values), line_style, marker in zip(Q_time_series.items(), line_styles, markers):
762     plt.plot(
763         time_steps,
764         values,
765         label=f"Q(z={z})",
766         marker=marker,
767         linestyle=line_style,
768         color='black', # Ensure black-and-white compliance
769         linewidth=2,
770         markersize=6,
771     )
772
773 # Add labels, title, legend, and grid
774 plt.xlabel("Time Steps", fontsize=12)
775 plt.ylabel("Q(z)", fontsize=12)
776 plt.title("Temporal Trends of Q-Function for Specific Thresholds", fontsize=14)
777 plt.grid(True, linestyle="--", linewidth=0.5)
778 plt.legend(fontsize=10, loc='upper left')
779
780 # Save and display the plot
781 plt.tight_layout()
782 plt.savefig("Temporal_Trends_of_Q_Function_BW.png", dpi=300)
783 plt.show()
784
785
786 # Simulated Q-function and G-function values
787 thresholds = np.linspace(1, 100, 100)
788 Q_values = [np.log10(z) * z if z < 50 else z ** 0.8 for z in thresholds]
789 G_values = [z ** 0.5 for z in thresholds] # Example G-function trend
790
791 # Overlay Plot
792 plt.figure(figsize=(10, 6))
793 plt.plot(thresholds, Q_values, label="Q-function", color='black', linestyle =
"dotted", linewidth=2)
794 plt.plot(thresholds, G_values, label="G-function", color='black', linestyle="--",
linewidth=2)
795 plt.xscale("log")
796 plt.yscale("log")
797 plt.xlabel("Threshold (z) (Log Scale)", fontsize=12)
798 plt.ylabel("Value (Log Scale)", fontsize=12)
799 plt.title("Comparison of Q-Function and G-Function Trends", fontsize=14)
800 plt.grid(True, which="both", linestyle="--", linewidth=0.5)
801 plt.legend()

```

```

802 plt.tight_layout()
803 plt.savefig("Q-function vs Q-function trends.png")
804 plt.show()
805
806
807 def map_rare_events_to_context(data, rare_events):
808     """
809     Map rare events back to their original context in the dataset.
810
811     Parameters:
812     - data: List of original texts (e.g., user reviews or feedback).
813     - rare_events: List of rare tokens.
814
815     Returns:
816     - rare_event_contexts: Dictionary mapping rare tokens to their contexts.
817     """
818     rare_event_contexts = {event: [] for event in rare_events}
819     for event in rare_events:
820         for text in data:
821             if event in text:
822                 rare_event_contexts[event].append(text)
823     return rare_event_contexts
824
825 # Example usage
826 rare_event_contexts = map_rare_events_to_context(train_df["text_prepro"], rare_events)
827 for event, contexts in rare_event_contexts.items():
828     print(f"Rare Event: {event}")
829     print("Contexts:", contexts)
830
831
832
833 def visualize_rare_token_q_contributions(rare_events, q_values):
834     """
835     Visualize Q-function contributions for rare events.
836
837     Parameters:
838     - rare_events: List of rare tokens.
839     - q_values: Q-function values corresponding to thresholds.
840     """
841     rare_q_values = [q_values[i] for i, token in enumerate(rare_events)]
842     plt.figure(figsize=(12, 6))
843     plt.bar(rare_events, rare_q_values, color="black")
844     plt.xlabel("Rare Tokens")
845     plt.ylabel("Q(z)")
846     plt.title("Q-Function Contributions for Rare Tokens")
847     plt.xticks(rotation=45)
848     plt.grid(True)
849     plt.show()
850
851 # Example usage
852 visualize_rare_token_q_contributions(rare_events, q_values)
853
854
855
856 def extract_contexts(text_data, rare_event, window=3):
857     """
858     Extract contexts for a rare event within a given window size.
859
860     Parameters:
861     - text_data: List of tokenized texts (sentences or documents).
862     - rare_event: The rare event (word) to find contexts for.
863     - window: Number of words before and after the rare event to include in the context.
864
865     Returns:
866     - contexts: List of contexts (substrings or word windows) where the rare event
867       occurs.
868     """
869     contexts = []
870     for text in text_data:

```

```

870         words = text.split() # Assuming `text_data` is tokenized
871         for i, word in enumerate(words):
872             if word == rare_event:
873                 start = max(0, i - window)
874                 end = min(len(words), i + window + 1)
875                 contexts.append(" ".join(words[start:end]))
876     return contexts
877
878
879
880 # Example text data
881 text_data = ["word1 is an example", "word2 appears here", "word1 and word3 are rare"]
882
883 # Extract contexts for a rare event
884 rare_event = "word1"
885 contexts = extract_contexts(text_data, rare_event, window=2)
886 print("Contexts for", rare_event, ":", contexts)
887
888
889 import numpy as np
890 import matplotlib.pyplot as plt
891 from scipy.stats import poisson
892
893 # Generate synthetic data for demonstration
894 np.random.seed(42)
895 frequencies = np.random.poisson(lam=5, size=1000) # Simulated token frequencies
896
897 # Define empirical G-function (cumulative contribution above threshold z)
898 def empirical_g_function(frequencies, z):
899     """Compute the empirical G-function for threshold z."""
900     return sum(freq for freq in frequencies if freq > z)
901
902 # Define theoretical C-function
903 def theoretical_c_function(z, Q_values):
904     """
905     Compute the theoretical C-function based on the Q-function.
906     Q_values should represent cumulative contributions (e.g., from rare events).
907     """
908     # Approximation using summation for Q-function values
909     C = sum((poisson.sf(k=z, mu=q) * q for q in Q_values))
910     return C
911
912 # Simulate Q-function values (cumulative contributions from rare events)
913 Q_values = np.linspace(0.1, 50, 100) # Rare-event contributions
914
915 # Define thresholds
916 thresholds = np.linspace(0, max(frequencies), 50)
917
918 # Compute empirical G-function and theoretical C-function for each threshold
919 empirical_g = [empirical_g_function(frequencies, z) for z in thresholds]
920 theoretical_c = [theoretical_c_function(z, Q_values) for z in thresholds]
921
922 # Visualization of Convergence
923 plt.figure(figsize=(12, 6))
924 plt.plot(thresholds, empirical_g, label="Empirical G-function", marker="o",
925          linestyle="-", color="black")
926 plt.plot(thresholds, theoretical_c, label="Theoretical C-function", marker="x",
927          linestyle="--", color="black")
928 plt.xlabel("Threshold (z)", fontsize=12)
929 plt.ylabel("Cumulative Contribution", fontsize=12)
930 plt.title("Convergence of G-function to C-function", fontsize=14)
931 plt.legend(fontsize=12)
932 plt.grid(True)
933 plt.tight_layout()
934 plt.savefig("convergence of G-function to C-function.png")
935 plt.show()
936
937 # Analyze the difference (convergence behavior)
938 convergence_difference = np.abs(np.array(empirical_g) - np.array(theoretical_c))

```

```

937
938 # Visualize the difference
939 plt.figure(figsize=(12, 6))
940 plt.plot(thresholds, convergence_difference, label="Difference (|Empirical G -
Theoretical C|)", color="black")
941 plt.xlabel("Threshold (z)", fontsize=12)
942 plt.ylabel("Difference", fontsize=12)
943 plt.title("Convergence Difference Between Empirical G and Theoretical C", fontsize=14)
944 plt.legend(fontsize=12)
945 plt.grid(True)
946 plt.tight_layout()
947 plt.savefig("convergence difference between G-function.png")
948 plt.show()
949
950
951
952 import numpy as np
953 import matplotlib.pyplot as plt
954
955 # Simulated data for empirical G-function and theoretical C-function
956 z_values = np.arange(0, 12, 1) # Threshold values (z)
957 empirical_G_values = np.random.randint(4000, 5000, len(z_values)) # Simulated empirical
G-function
958 theoretical_C_values = np.full(len(z_values), 3000) # Simulated theoretical C-function
as constant
959
960 # Model Calibration Function
961 def calibrate_model(empirical_G, theoretical_C, weights=None):
962     """
963     Calibrate the empirical G-function to better align with the theoretical C-function.
964     """
965     if weights is None:
966         weights = np.ones_like(empirical_G) # Default equal weighting
967
968     calibration_factor = (empirical_G - theoretical_C) * weights
969     calibrated_G = empirical_G - calibration_factor
970
971     return calibrated_G
972
973 # Apply model calibration
974 weights = np.linspace(1, 0.1, len(z_values)) # Example: higher weight for lower
thresholds
975 calibrated_G = calibrate_model(empirical_G_values, theoretical_C_values, weights=weights)
976
977 # Plot calibrated G-function against theoretical C-function
978 plt.figure(figsize=(10, 5))
979 plt.plot(z_values, calibrated_G, label="Calibrated G-function", color="black", linestyle
= "dotted")
980 plt.plot(z_values, theoretical_C_values, label="Theoretical C-function", color="black",
linestyle="-.")
981 plt.xlabel("Threshold (z)")
982 plt.ylabel("Cumulative Contribution")
983 plt.title("Calibrated G-function vs Theoretical C-function")
984 plt.legend()
985 plt.grid(True)
986 plt.savefig("calibrated G-functions vs Theoretical C-function.png")
987 plt.show()
988
989
990
991 import numpy as np
992 import matplotlib.pyplot as plt
993
994 # Simulate temporal evolution of G_f(z) and C(z)
995 time_steps = np.arange(1, 11) # Example: 10 time steps
996 z_values = np.linspace(1, 50, 50) # Threshold values (example)
997 empirical_G_values = np.exp(-z_values / 10) # Simulated initial G-values
998 theoretical_C_values = np.exp(-z_values / 20) # Simulated baseline C-values
999

```

```

1000 # Simulated growth over time
1001 temporal_G_values = [empirical_G_values * (1 + 0.1 * t) for t in time_steps] #
Simulated G-function growth
1002 temporal_C_values = [theoretical_C_values * (1 + 0.05 * t) for t in time_steps] #
Simulated C-function growth
1003
1004 # Visualization
1005 plt.figure(figsize=(10, 5))
1006
1007 # Define line styles and markers
1008 line_styles = ['solid', 'dotted', 'dashdot', 'dashed']
1009 markers = ['o', 's', 'd', 'x']
1010
1011 # Plot temporal G-function trends
1012 for t, G_values in enumerate(temporal_G_values, start=1):
1013     linestyle = line_styles[t % len(line_styles)] # Cycle through line styles
1014     marker = markers[t % len(markers)] # Cycle through markers
1015     plt.plot(
1016         z_values,
1017         G_values,
1018         label=f"G-function (t={t})",
1019         linestyle=linestyle,
1020         marker=marker,
1021         color='black',
1022         linewidth=1.5,
1023         markersize=5,
1024     )
1025
1026 # Plot baseline theoretical C-function
1027 plt.plot(
1028     z_values,
1029     theoretical_C_values,
1030     label="Theoretical C-function (Baseline)",
1031     color='black',
1032     linestyle='--',
1033     linewidth=2,
1034 )
1035
1036 # Add labels, title, legend, and grid
1037 plt.xlabel("Threshold (z)", fontsize=12)
1038 plt.ylabel("Cumulative Contribution", fontsize=12)
1039 plt.title("Temporal Evolution of G-function", fontsize=14)
1040 plt.grid(True, linestyle="--", linewidth=0.5)
1041 plt.legend(fontsize=10, loc='upper left')
1042
1043 # Save and display the plot
1044 plt.tight_layout()
1045 plt.savefig("temporal_evolution_of_G_function_BW.png", dpi=300)
1046 plt.show()
1047
1048
1049
1050 import numpy as np
1051 import matplotlib.pyplot as plt
1052
1053 # Threshold values (z)
1054 z_values = np.arange(0, 12, 1)
1055
1056 # Simulated data for empirical G-function for multiple datasets
1057 dataset1_G = np.random.randint(4000, 5000, len(z_values)) # Simulated dataset 1
1058 dataset2_G = np.random.randint(3000, 4500, len(z_values)) # Simulated dataset 2
1059 dataset3_G = np.random.randint(2000, 4000, len(z_values)) # Simulated dataset 3
1060
1061 # Prepare datasets for plotting
1062 datasets = {
1063     "Dataset 1": dataset1_G,
1064     "Dataset 2": dataset2_G,
1065     "Dataset 3": dataset3_G,
1066 }

```



```

1067
1068 # Define line styles and markers for distinction
1069 line_styles = ["solid", "dashed", "dotted"]
1070 markers = ["o", "s", "d"]
1071
1072 # Plot empirical G-functions for each dataset
1073 plt.figure(figsize=(10, 6))
1074
1075 for (name, G_values), linestyle, marker in zip(datasets.items(), line_styles, markers):
1076     plt.plot(
1077         z_values,
1078         G_values,
1079         label=name,
1080         linestyle=linestyle,
1081         marker=marker,
1082         color="black", # Use black for all lines
1083         linewidth=1.5,
1084         markersize=6,
1085     )
1086
1087 # Plot theoretical C-function
1088 theoretical_C_values = np.full(len(z_values), 3000) # Simulated theoretical C-function
1089 as constant
1090 plt.plot(
1091     z_values,
1092     theoretical_C_values,
1093     label="Theoretical C-function",
1094     linestyle="dashdot", # Unique style for theoretical C-function
1095     color="black",
1096     linewidth=2,
1097 )
1098
1099 # Add labels, legend, and grid
1100 plt.xlabel("Threshold (z)", fontsize=12)
1101 plt.ylabel("Cumulative Contribution", fontsize=12)
1102 plt.title("Comparison of Empirical G-function Across Datasets", fontsize=14)
1103 plt.legend(fontsize=10, loc="upper right")
1104 plt.grid(True, linestyle="--", linewidth=0.5)
1105
1106 # Save and display the plot
1107 plt.tight_layout()
1108 plt.savefig("Comparison_of_Empirical_G_Function_BW.png", dpi=300)
1109 plt.show()
1110
1111 # Identify changes in rare event contributions
1112 def identify_trends(temporal_G_values, threshold_z):
1113     trends = []
1114     for t, G_values in enumerate(temporal_G_values):
1115         contribution = G_values[threshold_z]
1116         trends.append((t, contribution))
1117     return trends
1118
1119 trends = identify_trends(temporal_G_values, threshold_z=5)
1120 print("Trends in rare event contributions over time:", trends)
1121
1122
1123
1124 import numpy as np
1125 import matplotlib.pyplot as plt
1126 from collections import Counter
1127
1128 # Example: Token frequency calculation
1129 tokens = [word for text in train_df["text_prepro"] for word in text.split()]
1130 token_counts = Counter(tokens)
1131
1132 # Sort by frequency
1133 sorted_token_counts = sorted(token_counts.values(), reverse=True)
1134

```

```

1135 # Plot frequency distribution
1136 plt.figure(figsize=(10, 6))
1137 plt.loglog(range(1, len(sorted_token_counts) + 1), sorted_token_counts, color =
"black",marker='o', linestyle='-')
1138 plt.title("Log-Log Plot of Token Frequency Distribution")
1139 plt.xlabel("Rank (log scale)")
1140 plt.ylabel("Frequency (log scale)")
1141 plt.grid(True, which="both", linestyle="--", linewidth=0.5)
1142 plt.savefig("Log-Log Plot of Token Frequency Distribution.png")
1143 plt.show()
1144
1145
1146 import numpy as np
1147 import matplotlib.pyplot as plt
1148
1149 # Simulated sorted token counts (replace with your actual data)
1150 sorted_token_counts = np.random.zipf(2, 100) # Simulated token frequencies
1151
1152 # Rank-Frequency Product
1153 ranks = np.arange(1, len(sorted_token_counts) + 1)
1154 frequencies = np.array(sorted_token_counts)
1155 rank_freq_product = ranks * frequencies
1156
1157 # Plot observed vs. theoretical
1158 plt.figure(figsize=(10, 6))
1159
1160 # Observed data
1161 plt.loglog(
1162     ranks,
1163     frequencies,
1164     label="Observed",
1165     linestyle="solid",
1166     color="black",
1167     linewidth=1.5,
1168 )
1169
1170 # Theoretical curve
1171 plt.loglog(
1172     ranks,
1173     1 / ranks,
1174     label="Theoretical (1/r)",
1175     linestyle="dashed",
1176     color="black",
1177     linewidth=1.5,
1178 )
1179
1180 # Add title, labels, legend, and grid
1181 plt.title("Validation of Zipf's Law", fontsize=14)
1182 plt.xlabel("Rank (log scale)", fontsize=12)
1183 plt.ylabel("Frequency (log scale)", fontsize=12)
1184 plt.legend(fontsize=10, loc="upper right")
1185 plt.grid(True, which="both", linestyle="--", linewidth=0.5)
1186
1187 # Save and display the plot
1188 plt.tight_layout()
1189 plt.savefig("Validation_of_Zipfs_Law_BW.png", dpi=300)
1190 plt.show()
1191
1192
1193
1194 # Cumulative Distribution Function
1195 cumulative_frequencies = np.cumsum(frequencies) / sum(frequencies)
1196
1197 # Plot CDF
1198 plt.figure(figsize=(10, 6))
1199 plt.plot(ranks, cumulative_frequencies, color = "black", linestyle =
"solid",label="Cumulative Frequency")
1200 plt.axhline(0.9, color='black', linestyle='--', label="90% Threshold")
1201 plt.title("Cumulative Distribution Function")

```

```

1202 plt.xlabel("Rank")
1203 plt.ylabel("Cumulative Frequency")
1204 plt.legend()
1205 plt.grid(True)
1206 plt.savefig("Cumulative Distribution Function.png")
1207 plt.show()
1208
1209 # Percentage Contribution of Rare Events (e.g., bottom 10%)
1210 tail_threshold = int(len(sorted_token_counts) * 0.9) # Bottom 10% ranks
1211 tail_contribution = cumulative_frequencies[tail_threshold]
1212 print(f"Contribution of the rare events (tail): {tail_contribution:.2%}")
1213
1214
1215
1216 from scipy.stats import powerlaw
1217
1218 # Fit Power-Law Distribution
1219 a, loc, scale = powerlaw.fit(frequencies, floc=0)
1220 theoretical_freq = powerlaw.pdf(ranks, a, loc, scale)
1221
1222 # Plot Observed vs. Fitted
1223 plt.figure(figsize=(10, 6))
1224 plt.loglog(ranks, frequencies, label="Observed", color = "black")
1225 plt.loglog(ranks, theoretical_freq, label="Fitted Power-Law", color = "black",
1226           linestyle='--')
1227 plt.title("Power-Law Fit to Frequency Distribution")
1228 plt.xlabel("Rank (log scale)")
1229 plt.ylabel("Frequency (log scale)")
1230 plt.legend()
1231 plt.grid(True, which="both", linestyle="--", linewidth=0.5)
1232 plt.savefig("Power-Law Fit to Frequency Distribution.png")
1233
1234 from sklearn.feature_extraction.text import TfidfVectorizer
1235 from sklearn.cluster import KMeans
1236
1237 # --- Semantic Clustering: TF-IDF and KMeans Clustering ---
1238 # Prepare the data for clustering
1239 vectorizer = TfidfVectorizer()
1240 X = vectorizer.fit_transform(rare_words_df["Word"])
1241
1242 # Apply KMeans clustering
1243 num_clusters = 5 # Specify the number of clusters
1244 kmeans = KMeans(n_clusters=num_clusters, random_state=42)
1245 kmeans.fit(X)
1246
1247 # Add cluster labels to the rare_words_df
1248 rare_words_df["Cluster"] = kmeans.labels_
1249
1250 # Visualize the clusters
1251 plt.figure(figsize=(10, 6))
1252 for cluster_id in range(num_clusters):
1253     cluster_words = rare_words_df[rare_words_df["Cluster"] == cluster_id]["Word"]
1254     plt.bar(cluster_words, [1] * len(cluster_words), label=f"Cluster {cluster_id}")
1255 plt.xticks(rotation=90)
1256 plt.xlabel("Words")
1257 plt.ylabel("Cluster Indicator")
1258 plt.title("Rare Word Clustering")
1259 plt.legend()
1260 plt.tight_layout()
1261 plt.show()
1262
1263
1264
1265 import matplotlib.pyplot as plt
1266 from wordcloud import WordCloud
1267
1268 # Word cloud for each cluster
1269 for cluster_id in range(num_clusters):

```

```

1270     cluster_words = rare_words_df[rare_words_df["Cluster"] == cluster_id]
1271     word_freq = dict(zip(cluster_words["Word"], cluster_words["Frequency"]))
1272
1273     # Generate word cloud
1274     wordcloud = WordCloud(
1275         width=800, height=400, background_color='white'
1276     ).generate_from_frequencies(word_freq)
1277
1278     # Plot the word cloud
1279     plt.figure(figsize=(10, 5))
1280     plt.imshow(wordcloud, interpolation='bilinear')
1281     plt.title(f"Word Cloud for Cluster {cluster_id}")
1282     plt.savefig("wordcloud.png")
1283     plt.axis("off")
1284     plt.show()
1285
1286
1287
1288     import numpy as np
1289     import matplotlib.pyplot as plt
1290
1291     # Example data: Replace this with actual frequency and rank data
1292     ranks = np.arange(1, 10001) # Ranks
1293     frequencies = 1 / (ranks ** 1.2) # Example power-law distribution
1294
1295     plt.figure(figsize=(10, 6))
1296     plt.loglog(ranks, frequencies, marker="o", linestyle="none", color = "black",
1297         label="Observed Data")
1298     plt.xlabel("Rank (log scale)")
1299     plt.ylabel("Frequency (log scale)")
1300     plt.title("Log-Log Plot of Token Frequency Distribution")
1301     plt.legend()
1302     plt.grid(True, which="both", linestyle="--")
1303     plt.savefig("Log-Log Plot of Token Frequency Distribution.png")
1304     plt.show()
1305
1306
1307     # Example G_f(z) and C(z) values (replace with actual calculations)
1308     z_values = np.arange(1, 20) # Thresholds
1309     empirical_G = 5000 / z_values # Replace with actual G_f(z) computation
1310     theoretical_C = 3000 / z_values # Replace with actual C(z)
1311
1312     difference = np.abs(empirical_G - theoretical_C)
1313
1314     plt.figure(figsize=(10, 6))
1315     plt.plot(z_values, empirical_G, label="Empirical G_f(z)", color="black")
1316     plt.plot(z_values, theoretical_C, label="Theoretical C(z)", linestyle="--",
1317         color="black")
1318     plt.plot(z_values, difference, label="Difference", linestyle=":", color="black")
1319     plt.xlabel("Threshold (z)")
1320     plt.ylabel("Cumulative Contribution")
1321     plt.title("Convergence Analysis: Empirical G_f(z) vs Theoretical C(z)")
1322     plt.legend()
1323     plt.grid()
1324     plt.savefig("Convergence Analysis: Empirical G_f(z) vs Theoretical C(z).png")
1325     plt.show()
1326
1327
1328     from powerlaw import Fit
1329
1330     # Example data: Replace with actual token frequencies
1331     data = np.random.zipf(a=1.5, size=1000) # Example Zipf distribution
1332     fit = Fit(data)
1333
1334     print(f"Alpha (Scaling Parameter): {fit.alpha}")
1335     print(f"KS Test Statistic: {fit.D}")
1336     print(f"P-value: {fit.power_law.D}")

```

```

1337
1338
1339
1340 # Variance of contributions from rare events
1341 rare_event_contributions = empirical_G - theoretical_C
1342 variance = np.var(rare_event_contributions)
1343
1344 print(f"Variance of Rare Event Contributions: {variance}")
1345
1346
1347 import numpy as np
1348 import matplotlib.pyplot as plt
1349
1350 # Simulate data
1351 z_values = np.linspace(1, 50, 50) # Threshold values
1352 time_steps = np.arange(1, 11) # Example time steps
1353 empirical_G = np.exp(-z_values / 10) # Simulated empirical G-function
1354 rare_event_contributions_over_time = [
1355     empirical_G / t for t in time_steps # Simulate decreasing contributions over time
1356 ]
1357
1358 # Define line styles and markers for distinction
1359 line_styles = ["solid", "dashed", "dotted", "dashdot"]
1360 markers = ["o", "s", "d", "x"]
1361
1362 # Create plot
1363 plt.figure(figsize=(10, 6))
1364
1365 for t, contributions in zip(time_steps, rare_event_contributions_over_time):
1366     linestyle = line_styles[t % len(line_styles)] # Cycle through line styles
1367     marker = markers[t % len(markers)] # Cycle through markers
1368     plt.plot(
1369         z_values,
1370         contributions,
1371         label=f"Time Step {t}",
1372         linestyle=linestyle,
1373         marker=marker,
1374         color="black", # Black for all lines
1375         linewidth=1.5,
1376         markersize=5,
1377     )
1378
1379 # Add labels, title, legend, and grid
1380 plt.xlabel("Threshold (z)", fontsize=12)
1381 plt.ylabel("Rare Event Contributions", fontsize=12)
1382 plt.title("Temporal Analysis of Rare Event Contributions", fontsize=14)
1383 plt.grid(True, linestyle="--", linewidth=0.5)
1384 plt.legend(fontsize=10, loc="upper right")
1385
1386 # Save and display the plot
1387 plt.tight_layout()
1388 plt.savefig("Temporal_Analysis_of_Rare_Event_Contributions_BW.png", dpi=300)
1389 plt.show()
1390
1391
1392 from scipy.optimize import curve_fit
1393
1394 # Function for fitting a power-law
1395 def power_law(x, a, b):
1396     return b * (x ** -a)
1397
1398 # Fit power-law to data
1399 params, _ = curve_fit(power_law, ranks, frequencies)
1400 alpha = params[0]
1401 print(f"Tail Index (Alpha): {alpha}")
1402
1403
1404
1405 from numpy.linalg import eigvalsh

```

```

1406
1407 # Example covariance matrix
1408 cov_matrix = np.random.rand(10, 10)
1409 eigenvalues = eigvalsh(cov_matrix)
1410
1411 plt.figure(figsize=(10, 6))
1412 plt.plot(np.sort(eigenvalues), marker="o", label="Eigenvalues")
1413 plt.xlabel("Index")
1414 plt.ylabel("Eigenvalue")
1415 plt.title("Spectral Analysis: Eigenvalues of Covariance Matrix")
1416 plt.legend()
1417 plt.grid()
1418 plt.savefig("Spectral Analysis: Eigenvalues of Covariance Matrix.png")
1419 plt.show()
1420
1421
1422
1423 # Example tail contributions
1424 tail_contributions = np.cumsum(frequencies[::-1]) / np.sum(frequencies)
1425
1426 plt.figure(figsize=(10, 6))
1427 plt.plot(ranks[::-1], tail_contributions, label="Tail Contribution", color = "black",
1428         linestyle = "dotted")
1429 plt.axhline(0.9, color="black", linestyle="--", label="90% Contribution Threshold")
1430 plt.xlabel("Rank (Descending Order)")
1431 plt.ylabel("Cumulative Contribution")
1432 plt.title("Tail Contribution Analysis")
1433 plt.legend()
1434 plt.grid()
1435 plt.savefig("Tail Contribution Analysis.png")
1436 plt.show()
1437
1438
1439 # Define a threshold for residuals (e.g., 1% of average vocabulary size)
1440 a_threshold = 0.01 * np.mean(actual_vocab_sizes) # 1% of the mean vocabulary size
1441
1442 # Residual difference between observed and LNRE (Heap's Law)
1443 residuals = np.abs(np.array(actual_vocab_sizes) - np.array(lnre_vocab))
1444
1445 # Check convergence
1446 if np.max(residuals) < a_threshold:
1447     print("Vocabulary growth stabilizes (converges globally).")
1448 else:
1449     print("Vocabulary growth indicates divergence.")
1450
1451
1452
1453 import numpy as np
1454 import matplotlib.pyplot as plt
1455 from collections import Counter
1456
1457 # ----- Functions -----
1458
1459 # Tokenize function (example implementation)
1460 def tokenize(text):
1461     """Basic tokenizer splitting by whitespace."""
1462     return text.split()
1463
1464 # Frequency distribution calculation
1465 def calculate_frequency_distribution(tokens):
1466     """Calculate the frequency distribution of tokens."""
1467     token_counts = Counter(tokens)
1468     total_count = sum(token_counts.values())
1469     return {token: count / total_count for token, count in token_counts.items()}
1470
1471 # Vocabulary calculation
1472 def calculate_vocabulary(tokens, N):
1473     """Calculate vocabulary size for the first N tokens."""

```

```

1474         return len(set(tokens[:N]))
1475
1476 # Rare event frequency calculation
1477 def calculate_rare_event_frequencies(tokens, threshold=0.01):
1478     """
1479     Identify rare events based on a frequency threshold.
1480
1481     Parameters:
1482     - tokens: List of tokens
1483     - threshold: Frequency threshold for defining rare events
1484
1485     Returns:
1486     - List of rare event frequencies
1487     """
1488     freq_dist = calculate_frequency_distribution(tokens)
1489     rare_events = {token: freq for token, freq in freq_dist.items() if freq < threshold}
1490     return list(rare_events.values())
1491
1492 # Rank calculation (Zipf's law)
1493 def rank(tokens):
1494     """Calculate rank of tokens based on their frequency."""
1495     freq_dist = Counter(tokens)
1496     return sorted(freq_dist.values(), reverse=True)
1497
1498 # Rare event contributions
1499 def calculate_rare_event_contributions(tokens, N, threshold=0.01):
1500     """
1501     Calculate the contributions of rare events for the first N tokens.
1502
1503     Parameters:
1504     - tokens: List of tokens
1505     - N: Number of tokens to consider
1506     - threshold: Frequency threshold for rare events
1507
1508     Returns:
1509     - Contribution of rare events
1510     """
1511     token_counts = Counter(tokens[:N])
1512     total_tokens = sum(token_counts.values())
1513     rare_events = {token: count for token, count in token_counts.items() if count /
1514                    total_tokens < threshold}
1515     rare_contribution = sum(rare_events.values()) / total_tokens
1516     return rare_contribution
1517
1518 # Vocabulary growth models
1519 def heaps_law(N, k=20, beta=0.6):
1520     """Vocabulary size based on Heap's law (LNRE)."""
1521     return k * (N ** beta)
1522
1523 def lstm_vocab_growth(N):
1524     """
1525     Simulate LSTM vocabulary growth.
1526     This assumes diminishing returns due to training bias toward frequent words.
1527     """
1528     return np.log(N) ** 2
1529
1530 def bert_vocab_growth(N, max_vocab=30000):
1531     """
1532     Simulate BERT vocabulary growth.
1533     BERT relies on subword tokenization, so the vocabulary growth saturates early.
1534     """
1535     return max_vocab * (1 - np.exp(-N / max_vocab))
1536
1537 # ----- Data and Analysis -----
1538
1539 # Example tokens (replace with actual dataset)
1540 tokens = ["word" + str(i) for i in range(1, 10001)] * 5 # Example dataset with repeated
tokens

```

```

1541 # Token counts for analysis
1542 N_values = np.logspace(3, 6, num=50, dtype=int) # Token counts from 1,000 to 1,000,000
1543
1544 # Actual vocabulary sizes from the dataset
1545 actual_vocab_sizes = [calculate_vocabulary(tokens, N) for N in N_values]
1546
1547 # Rare event contributions
1548 rare_event_contributions = np.array([calculate_rare_event_contributions(tokens, N) for N
in N_values])
1549
1550 # Simulate LNRE (Heap's Law), LSTM, and BERT growth for comparison
1551 lnre_vocab = [heaps_law(N) for N in N_values]
1552 lstm_vocab = [lstm_vocab_growth(N) for N in N_values]
1553 bert_vocab = [bert_vocab_growth(N) for N in N_values]
1554
1555 # ----- Plots -----
1556
1557 # 1. Vocabulary Growth Comparison
1558 plt.figure(figsize=(12, 6))
1559 plt.plot(N_values, lnre_vocab, label="LNRE (Heap's Law)", color="blue", linewidth=2)
1560 plt.plot(N_values, lstm_vocab, label="LSTM", color="red", linestyle="--", linewidth=2)
1561 plt.plot(N_values, bert_vocab, label="BERT", color="green", linestyle=":", linewidth=2)
1562 plt.plot(N_values, actual_vocab_sizes, label="Dataset (Actual)", color="purple",
linewidth=2)
1563
1564 # Log scale for better visualization
1565 plt.xscale("log")
1566 plt.yscale("log")
1567 plt.xlabel("Number of Tokens (N)", fontsize=12)
1568 plt.ylabel("Vocabulary Size (V)", fontsize=12)
1569 plt.title("Vocabulary Growth Comparison: LNRE vs LSTM vs BERT", fontsize=14)
1570 plt.legend(fontsize=12)
1571 plt.grid(True, which="both", linestyle="--", linewidth=0.5)
1572 plt.tight_layout()
1573 plt.savefig("vocabulary_growth_comparison.png", dpi=300)
1574 plt.show()
1575
1576 # 2. Rare Event Contributions
1577 plt.figure(figsize=(12, 6))
1578 plt.plot(N_values, rare_event_contributions, label="Rare Event Contributions",
color="orange", linewidth=2)
1579 plt.xscale("log")
1580 plt.xlabel("Number of Tokens (N)", fontsize=12)
1581 plt.ylabel("Rare Event Contribution", fontsize=12)
1582 plt.title("Rare Event Contributions Over Tokens", fontsize=14)
1583 plt.legend(fontsize=12)
1584 plt.grid(True, which="both", linestyle="--", linewidth=0.5)
1585 plt.tight_layout()
1586 plt.savefig("rare_event_contributions.png", dpi=300)
1587 plt.show()
1588
1589 # ----- Convergence Analysis -----
1590
1591 # Residual difference between observed and LNRE (Heap's Law)
1592 residuals = np.abs(np.array(actual_vocab_sizes) - np.array(lnre_vocab))
1593 a_threshold = 0.05 # Example threshold for convergence
1594 if np.max(residuals) < a_threshold:
1595     print("Vocabulary growth stabilizes (converges globally).")
1596 else:
1597     print("Vocabulary growth fluctuates (diverges globally).")
1598
1599 # Variance reduction for rare events
1600 variance_reduction = np.var(rare_event_contributions, axis=0)
1601 if np.max(np.abs(np.diff(variance_reduction))) < 1e-5: # Small threshold for
stabilization
1602     print("Rare event contributions stabilize, indicating global convergence.")
1603 else:
1604     print("Rare event contributions fluctuate, indicating divergence.")
1605

```



```

1606
1607
1608 # Variance reduction for rare events
1609 variance_reduction = np.var(rare_event_contributions) # Variance is a scalar in this
case
1610
1611 # Check for stabilization
1612 if variance_reduction < 1e-5: # Small threshold for stabilization
1613     print("Rare event contributions stabilize, indicating global convergence.")
1614 else:
1615     print("Rare event contributions fluctuate, indicating divergence.")
1616
1617
1618
1619 import numpy as np
1620 import pandas as pd
1621 import matplotlib.pyplot as plt
1622 from collections import Counter
1623
1624 # Step 1: Extract token frequencies from train_df["text_prepro"]
1625 tokens = [word for text in train_df["text_prepro"].dropna() for word in text.split()]
1626 token_counts = Counter(tokens)
1627
1628 # Step 2: Sort tokens by frequency
1629 sorted_token_counts = sorted(token_counts.values(), reverse=True)
1630 rank = np.arange(1, len(sorted_token_counts) + 1) # Rank of tokens
1631
1632 # Step 3: Normalize demand distribution and calculate cumulative contribution
1633 demand = np.array(sorted_token_counts) / sum(sorted_token_counts) # Normalize demands
1634 cumulative_demand = np.cumsum(demand) # Cumulative contribution
1635
1636 # Step 4: Define the length of the tail
1637 tail_threshold = int(0.9 * len(sorted_token_counts)) # Bottom 90% of tokens
1638 tail_contribution = cumulative_demand[tail_threshold - 1] # Contribution from the tail
1639
1640 # Step 5: Plot the conceptual model
1641 plt.figure(figsize=(12, 8))
1642
1643 # Demand distribution (log-log scale)
1644 plt.subplot(2, 1, 1)
1645 plt.loglog(rank, demand, label="Demand per Token", color="blue")
1646 plt.axvline(x=tail_threshold, color="red", linestyle="--", label="Tail Threshold")
1647 plt.xlabel("Token Rank (Log Scale)")
1648 plt.ylabel("Demand per Token (Log Scale)")
1649 plt.title("Demand Distribution by Token Rank")
1650 plt.legend()
1651 plt.grid(True, which="both", linestyle="--", linewidth=0.5)
1652
1653 # Cumulative contribution
1654 plt.subplot(2, 1, 2)
1655 plt.plot(rank, cumulative_demand, label="Cumulative Contribution", color="green")
1656 plt.axvline(x=tail_threshold, color="red", linestyle="--", label="Tail Threshold")
1657 plt.axhline(y=tail_contribution, color="purple", linestyle="--", label=f"Tail
Contribution ({tail_contribution:.1%})")
1658 plt.xlabel("Token Rank")
1659 plt.ylabel("Cumulative Contribution")
1660 plt.title("Cumulative Contribution of Tokens")
1661 plt.legend()
1662 plt.grid(True, linestyle="--", linewidth=0.5)
1663
1664 plt.tight_layout()
1665 plt.savefig("conceptual_model_token_visualization.png", dpi=300)
1666 plt.show()
1667
1668 # Print tail contribution for reference
1669 print(f"Contribution of the tail (bottom 90% tokens): {tail_contribution:.2%}")
1670
1671
1672

```

```

1673 import numpy as np
1674 import pandas as pd
1675 import matplotlib.pyplot as plt
1676 from collections import Counter
1677
1678 # Load data (assume train_df["text_prepro"] is already preprocessed)
1679 # Tokenize the preprocessed text
1680 tokens = [word for text in train_df["text_prepro"] for word in text.split()]
1681
1682 # Generate token frequencies
1683 token_frequencies = Counter(tokens)
1684 sorted_frequencies = np.array(sorted(token_frequencies.values(), reverse=True))
1685
1686
1687
1688 # Define Q-function for cumulative contributions
1689 def Q_function(z, frequencies):
1690     """Compute the cumulative contributions up to threshold z."""
1691     return np.sum(frequencies[frequencies <= z])
1692
1693 # Compute Q(z) for the dataset
1694 z_values = np.logspace(0, np.log10(max(sorted_frequencies)), num=50)
1695 Q_values = [Q_function(z, sorted_frequencies) for z in z_values]
1696
1697 # Tail contributions (e.g., bottom 10% frequencies)
1698 tail_threshold = np.percentile(sorted_frequencies, 10) # Bottom 10%
1699 Q_tail_values = [Q_function(z, sorted_frequencies[sorted_frequencies <= tail_threshold])
1700                  for z in z_values]
1701
1702 # Plot Q(z) and Q_tail(z)
1703 plt.figure(figsize=(10, 6))
1704 plt.plot(z_values, Q_values, label="Total Cumulative Contributions (Q(z))",
1705          color="black")
1706 plt.plot(z_values, Q_tail_values, label="Tail Contributions (Q_tail(z))",
1707          linestyle="--", color="gray")
1708 plt.xscale("log")
1709 plt.xlabel("Threshold (z)")
1710 plt.ylabel("Cumulative Contribution")
1711 plt.title("Uniform Partitioning: Q(z) vs. Q_tail(z)")
1712 plt.legend()
1713 plt.grid(True, which="both", linestyle="--", linewidth=0.5)
1714 plt.tight_layout()
1715 plt.savefig("Uniform Partitioning.png")
1716 plt.show()
1717
1718
1719 import numpy as np
1720 import pandas as pd
1721 import matplotlib.pyplot as plt
1722 from collections import Counter
1723
1724 # Load the dataset (ensure train_df is preloaded and contains "text_prepro")
1725 # Simulated train_df["text_prepro"]
1726 train_df = pd.DataFrame({"text_prepro": ["word1 word2 word3 word1 word2 word1", "word4
1727 word5 word1 word6"]})
1728
1729 # Step 1: Extract token frequencies
1730 tokens = []
1731 train_df["text_prepro"].dropna().apply(lambda x: tokens.extend(x.split()))
1732 token_counts = Counter(tokens)
1733 sorted_frequencies = np.array(sorted(token_counts.values(), reverse=True))
1734
1735 # Step 2: Define the density function
1736 def density_function(x, alpha=2.5):
1737     """Power-law density function."""
1738     return x ** -alpha if x > 0 else 0
1739
1740 # Step 3: Define G-function and Q-function

```

```

1738 def G_function(z, density_func):
1739     """G-function: Contribution of frequent events above z."""
1740     return sum(density_func(x) for x in range(int(z), len(sorted_frequencies) + 1))
1741
1742 def Q_function(z, density_func):
1743     """Q-function: Cumulative contribution of rare events below z."""
1744     return sum(x * density_func(x) for x in range(1, int(z) + 1))
1745
1746 # Step 4: Compute G(z) and Q(z) for the dataset
1747 z_values = np.linspace(1, len(sorted_frequencies), 50)
1748 G_values = [G_function(z, density_function) for z in z_values]
1749 Q_values = [Q_function(z, density_function) for z in z_values]
1750
1751 # Step 5: Normalize G and Q for comparison
1752 G_values_normalized = np.array(G_values) / max(G_values)
1753 Q_values_normalized = np.array(Q_values) / max(Q_values)
1754
1755 # Step 6: Plot results
1756 plt.figure(figsize=(10, 6))
1757 plt.plot(z_values, G_values_normalized, label="Normalized G(z) (Frequent Events)",
1758         color="black", linestyle="--")
1759 plt.plot(z_values, Q_values_normalized, label="Normalized Q(z) (Rare Events)",
1760         color="black", linestyle="-")
1761 plt.xlabel("Threshold (z)", fontsize=12)
1762 plt.ylabel("Normalized Contribution", fontsize=12)
1763 plt.title("Lemma 1: Contribution Concentration in Long-Tail Markets", fontsize=14)
1764 plt.grid(True, linestyle="--", linewidth=0.5)
1765 plt.legend(fontsize=12)
1766 plt.tight_layout()
1767 plt.savefig("lemmal1_concentration_long_tail.png", dpi=300)
1768 plt.show()
1769
1770 # Simulate time-varying density
1771 def density_over_time(x, t, alpha=1.5):
1772     """Simulate time-varying density with fluctuations."""
1773     return 1 / (x ** alpha) * (1 + 0.1 * np.sin(2 * np.pi * t / 10))
1774
1775 # Compute Q(z, t) over time
1776 time_steps = np.arange(1, 11) # 10 time steps
1777 z_values = np.logspace(1, np.log10(len(sorted_frequencies)), 50)
1778
1779 Q_time_series = [
1780     [Q_function(z, lambda x: density_over_time(x, t)) for z in z_values]
1781     for t in time_steps
1782 ]
1783
1784 # Temporal analysis visualization
1785 plt.figure(figsize=(10, 6))
1786 for t, Q_values in enumerate(Q_time_series):
1787     plt.plot(z_values, Q_values, label=f"Time Step {t+1}", color="black", linestyle="--"
1788             if t % 2 else "-")
1789 plt.xscale("log")
1790 plt.xlabel("Threshold (z)")
1791 plt.ylabel("Q(z)")
1792 plt.title("Temporal Evolution of Q-Function")
1793 plt.legend()
1794 plt.grid(True, which="both", linestyle="--", linewidth=0.5)
1795 plt.tight_layout()
1796 plt.savefig("Temporal Evolution of Q-Function.png")
1797 plt.show()
1798
1799
1800 import numpy as np
1801 import matplotlib.pyplot as plt
1802
1803 # Define density function (e.g., long-tail distribution)

```

```

1804 def density_function(x, alpha=2, k=1):
1805     return k * x ** -alpha if x >= 1 else 0
1806
1807 # Logarithmic Transformation for Q(z)
1808 def Q_log_transformed(z, density_func):
1809     u_values = np.linspace(np.log(1), np.log(z), 1000)
1810     return np.trapz([density_func(np.exp(u)) * np.exp(u) for u in u_values], u_values)
1811
1812 # Normalized Transformation for G(z)
1813 def G_normalized(z, density_func):
1814     v_values = np.linspace(0, 1, 1000)
1815     return z ** 2 * np.trapz([v * density_func(v * z) for v in v_values], v_values)
1816
1817 # Compute results
1818 z_values = np.logspace(1, 3, 50)
1819 Q_values_log = [Q_log_transformed(z, density_function) for z in z_values]
1820 G_values_normalized = [G_normalized(z, density_function) for z in z_values]
1821
1822 # Plot results
1823 plt.figure(figsize=(12, 6))
1824 plt.plot(z_values, Q_values_log, label="Log-Transformed Q(z)", color="black",
1825         linestyle="-")
1826 plt.plot(z_values, G_values_normalized, label="Normalized G(z)", color="black",
1827         linestyle="--")
1828 plt.xscale("log")
1829 plt.xlabel("Threshold (z)")
1830 plt.ylabel("Cumulative Contribution")
1831 plt.title("Variable Substitution in Q(z) and G(z)")
1832 plt.legend()
1833 plt.grid(True, which="both", linestyle="--", linewidth=0.5)
1834 plt.tight_layout()
1835 plt.savefig("substitution_analysis.png")
1836 plt.show()
1837
1838 import numpy as np
1839 import matplotlib.pyplot as plt
1840
1841 # Generate example frequency data
1842 z_max = 1000 # Maximum threshold
1843 n_partitions = 20 # Number of partitions
1844 z_values = np.linspace(1, z_max, 1000)
1845 frequencies = 1 / z_values # Example long-tail distribution
1846
1847 # Define Q(z) and G(z)
1848 def Q_function(z):
1849     return np.cumsum(frequencies[:z])
1850
1851 def G_function(z):
1852     return np.sum(frequencies[z:])
1853
1854 # Uniform partitioning
1855 partitions = np.linspace(1, z_max, n_partitions + 1)
1856 Q_contributions = []
1857 G_contributions = []
1858
1859 for i in range(1, len(partitions)):
1860     z_start = int(partitions[i - 1])
1861     z_end = int(partitions[i])
1862     Q_contributions.append(Q_function(z_end)[-1] - Q_function(z_start)[-1])
1863     G_contributions.append(G_function(z_end))
1864
1865 # Plot Q(z) and G(z) contributions
1866 plt.figure(figsize=(12, 6))
1867 plt.bar(range(1, n_partitions + 1), Q_contributions, label="Q(z) Contributions",
1868         alpha=0.7, color="black")
1869 plt.bar(range(1, n_partitions + 1), G_contributions, label="G(z) Contributions",
1870         alpha=0.7, hatch="//", color="gray")

```

```

1869 plt.xlabel("Partition Index")
1870 plt.ylabel("Cumulative Contribution")
1871 plt.title("Granular Contributions via Uniform Partitioning")
1872 plt.legend()
1873 plt.grid(True, linestyle="--", linewidth=0.5)
1874 plt.tight_layout()
1875 plt.savefig("granular_contributions_uniform_partition.png", dpi=300)
1876 plt.show()
1877
1878
1879
1880 import numpy as np
1881 import matplotlib.pyplot as plt
1882
1883 # Simulate relative frequencies over time
1884 time_points = [1, 2, 3, 4, 5] # Discrete time points
1885 N = 10 # Number of subintervals
1886 domain = np.linspace(0, 1, 1000) # Domain
1887
1888 # Simulate density function f_t(t) at each time point
1889 densities = {t: np.sin(2 * np.pi * domain * t) ** 2 + 0.1 for t in time_points} #
Example density
1890
1891 # Compute v_i,n,t for each subinterval and time
1892 subintervals = np.linspace(0, 1, N + 1)
1893 frequencies_over_time = {t: [] for t in time_points}
1894
1895 for t in time_points:
1896     for i in range(len(subintervals) - 1):
1897         sub_start, sub_end = subintervals[i], subintervals[i + 1]
1898         freq = np.trapz(densities[t][(domain >= sub_start) & (domain < sub_end)],
1899                        domain[(domain >= sub_start) & (domain < sub_end)])
1900         frequencies_over_time[t].append(freq)
1901
1902 # Plot 1: Visualize v_i,n,t over time
1903 plt.figure(figsize=(12, 6))
1904 for t in time_points:
1905     plt.plot(range(1, N + 1), frequencies_over_time[t], label=f"Time {t}",
1906             linestyle="--", color="black")
1907 plt.xlabel("Subinterval Index")
1908 plt.ylabel("Relative Frequency (v_i,n,t)")
1909 plt.title("Relative Frequencies Over Time (Emerging Trends)")
1910 plt.legend()
1911 plt.grid(color="gray", linestyle="--", linewidth=0.5)
1912 plt.savefig("relative_frequencies_over_time.png", dpi=300, bbox_inches="tight")
1913 plt.show()
1914
1915 # Identify Emerging Trends
1916 trend_changes = {i: [] for i in range(1, N + 1)}
1917 for i in range(N):
1918     for t in range(len(time_points) - 1):
1919         change = frequencies_over_time[time_points[t + 1]][i] -
frequencies_over_time[time_points[t]][i]
1920         trend_changes[i + 1].append(change)
1921
1922 # Plot 2: Visualize Trend Changes
1923 plt.figure(figsize=(12, 6))
1924 for i, changes in trend_changes.items():
1925     plt.plot(time_points[1:], changes, label=f"Subinterval {i}", linestyle="-",
1926             color="black")
1927 plt.xlabel("Time Points")
1928 plt.ylabel("Change in v_i,n,t")
1929 plt.title("Trend Changes in Relative Frequencies")
1930 plt.legend()
1931 plt.grid(color="gray", linestyle="--", linewidth=0.5)
1932 plt.savefig("trend_changes_relative_frequencies.png", dpi=300, bbox_inches="tight")
1933 plt.show()
1934
1935

```

```

1934 import numpy as np
1935 import matplotlib.pyplot as plt
1936
1937 # Define density function (e.g., long-tail distribution)
1938 def density_function(x, alpha=2, k=1):
1939     return k * x ** -alpha if x >= 1 else 0
1940
1941 # Logarithmic Transformation for Q(z)
1942 def Q_log_transformed(z, density_func):
1943     u_values = np.linspace(np.log(1), np.log(z), 1000)
1944     return np.trapz([density_func(np.exp(u)) * np.exp(u) for u in u_values], u_values)
1945
1946 # Normalized Transformation for G(z)
1947 def G_normalized(z, density_func):
1948     v_values = np.linspace(0, 1, 1000)
1949     return z ** 2 * np.trapz([v * density_func(v * z) for v in v_values], v_values)
1950
1951 # Compute results
1952 z_values = np.logspace(1, 3, 50)
1953 Q_values_log = [Q_log_transformed(z, density_function) for z in z_values]
1954 G_values_normalized = [G_normalized(z, density_function) for z in z_values]
1955
1956 # Plot results
1957 plt.figure(figsize=(12, 6))
1958 plt.plot(z_values, Q_values_log, label="Log-Transformed Q(z)", color="black",
1959          linestyle="-")
1960 plt.plot(z_values, G_values_normalized, label="Normalized G(z)", color="black",
1961          linestyle="--")
1962 plt.xscale("log")
1963 plt.xlabel("Threshold (z)")
1964 plt.ylabel("Cumulative Contribution")
1965 plt.title("Variable Substitution in Q(z) and G(z)")
1966 plt.legend()
1967 plt.grid(True, which="both", linestyle="--", linewidth=0.5)
1968 plt.tight_layout()
1969 plt.savefig("substitution_analysis.png")
1970 plt.show()
1971
1972 import numpy as np
1973 import matplotlib.pyplot as plt
1974
1975 # Define density function (non-increasing)
1976 def density_function(x, alpha=2, k=1):
1977     return k * x ** -alpha if x >= 1 else 0
1978
1979 # Variable transformations
1980 def Q_log_transformed(z, density_func):
1981     u_values = np.linspace(np.log(1), np.log(z), 1000)
1982     return np.trapz([density_func(np.exp(u)) * np.exp(u) for u in u_values], u_values)
1983
1984 def G_normalized(z, density_func):
1985     v_values = np.linspace(0, 1, 1000)
1986     return z * np.trapz([density_func(v * z) * v for v in v_values], v_values)
1987
1988 # Compute and plot results
1989 z_values = np.logspace(1, 3, 50)
1990 Q_values_log = [Q_log_transformed(z, density_function) for z in z_values]
1991 G_values_normalized = [G_normalized(z, density_function) for z in z_values]
1992
1993 plt.figure(figsize=(12, 6))
1994 plt.plot(z_values, Q_values_log, label="Log-Transformed Q(z)", color="black",
1995          linestyle="-")
1996 plt.plot(z_values, G_values_normalized, label="Normalized G(z)", color="black",
1997          linestyle="--")
1998 plt.xscale("log")
1999 plt.xlabel("Threshold (z)")
2000 plt.ylabel("Cumulative Contribution")

```

```

1999 plt.title("Variable Change in Q(z) and G(z) for Improved Tail Analysis")
2000 plt.legend()
2001 plt.grid(True, which="both", linestyle="--", linewidth=0.5)
2002 plt.tight_layout()
2003 plt.savefig("variable_change_analysis.png")
2004 plt.show()
2005
2006
2007
2008 # Define density function with inconsistency
2009 def density_function_inconsistent(x):
2010     return np.random.uniform(0.9, 1.1) * density_function_non_increasing(x)
2011
2012 # Compute Q(z) and Q_tail(z) for inconsistent density
2013 Q_values = [Q_function(z, lambda x: density_function_inconsistent(x)) for z in z_values]
2014 Q_tail_values = [Q_tail_function(z, lambda x: density_function_inconsistent(x)) for z in
2015 z_values]
2016
2017 # Plot results
2018 plt.figure(figsize=(12, 6))
2019 plt.plot(z_values, Q_values, label="Total Contributions (Q(z))", color="black",
2020 linewidth=2)
2021 plt.plot(z_values, Q_tail_values, label="Tail Contributions (Q_tail(z))", color="black",
2022 linestyle="--", linewidth=2)
2023 plt.xscale("log")
2024 plt.xlabel("Threshold (z)")
2025 plt.ylabel("Cumulative Contribution")
2026 plt.title("Inconsistent Density: Q(z) vs. Q_tail(z)")
2027 plt.legend()
2028 plt.grid(True, which="both", linestyle="--", linewidth=0.5)
2029 plt.tight_layout()
2030 plt.savefig("inconsistent_density_bw.png")
2031 plt.show()
2032
2033 # Define density function for non-increasing behavior
2034 def density_function_non_increasing(x, alpha=2, k=1):
2035     return k * x ** -alpha if x >= 1 else 0
2036
2037 # Compute Q(z) and Q_tail(z) for non-increasing density
2038 Q_values = [Q_function(z, lambda x: density_function_non_increasing(x)) for z in
2039 z_values]
2040 Q_tail_values = [Q_tail_function(z, lambda x: density_function_non_increasing(x)) for z
2041 in z_values]
2042
2043 # Plot results
2044 plt.figure(figsize=(12, 6))
2045 plt.plot(z_values, Q_values, label="Total Contributions (Q(z))", color="black",
2046 linewidth=2)
2047 plt.plot(z_values, Q_tail_values, label="Tail Contributions (Q_tail(z))", color="black",
2048 linestyle="--", linewidth=2)
2049 plt.xscale("log")
2050 plt.xlabel("Threshold (z)")
2051 plt.ylabel("Cumulative Contribution")
2052 plt.title("Non-Increasing Density: Q(z) vs. Q_tail(z)")
2053 plt.legend()
2054 plt.grid(True, which="both", linestyle="--", linewidth=0.5)
2055 plt.tight_layout()
2056 plt.savefig("non_increasing_density_bw.png")
2057 plt.show()
2058
2059
2060 import numpy as np
2061 import matplotlib.pyplot as plt
2062
2063 # Example dataset: token frequencies (replace with actual data)
2064 tokens = [word for text in train_df["text_prepro"] for word in text.split()]

```

```

2061 token_frequencies = dict(Counter(tokens))
2062
2063 # Sort frequencies in descending order
2064 sorted_frequencies = np.array(sorted(token_frequencies.values(), reverse=True))
2065
2066 # Define the density function
2067 def density_function(x, sorted_frequencies):
2068     return 1 / len(sorted_frequencies) if x in range(1, len(sorted_frequencies)+1) else 0
2069
2070 # Compute Q(z) and Q_tail(z)
2071 def Q_function(z, density_func):
2072     return sum(density_func(x, sorted_frequencies) for x in range(1, int(z)+1))
2073
2074 def Q_tail_function(z, density_func):
2075     return Q_function(len(sorted_frequencies), density_func) - Q_function(z,
        density_func)
2076
2077 # Compute values for Q(z) and Q_tail(z)
2078 z_values = np.logspace(1, np.log10(len(sorted_frequencies)), 50)
2079 Q_values = [Q_function(z, density_function) for z in z_values]
2080 Q_tail_values = [Q_tail_function(z, density_function) for z in z_values]
2081
2082 # Plot results
2083 plt.figure(figsize=(12, 6))
2084 plt.plot(z_values, Q_values, label="Total Cumulative Contributions (Q(z))",
        color="black", linewidth=2)
2085 plt.plot(z_values, Q_tail_values, label="Tail Contributions (Q_tail(z))", color="black",
        linestyle="--", linewidth=2)
2086 plt.xscale("log")
2087 plt.xlabel("Threshold (z)")
2088 plt.ylabel("Cumulative Contribution")
2089 plt.title("Uniform Partitioning: Q(z) vs. Q_tail(z)")
2090 plt.legend()
2091 plt.grid(True, which="both", linestyle="--", linewidth=0.5)
2092 plt.tight_layout()
2093 plt.savefig("uniform_partitioning_bw.png")
2094 plt.show()
2095
2096
2097
2098 import numpy as np
2099 import pandas as pd
2100 import matplotlib.pyplot as plt
2101 from collections import Counter
2102 from scipy.stats import powerlaw
2103 from scipy.optimize import curve_fit
2104 from sklearn.metrics import mean_squared_error
2105
2106 # Tokenize the text and calculate frequencies
2107 def tokenize_and_count(data):
2108     all_tokens = []
2109     data.dropna().apply(lambda text: all_tokens.extend(text.split()))
2110     return Counter(all_tokens)
2111
2112 # Plot Rank-Frequency on a Log-Log Scale
2113 def plot_log_log_rank_frequency(freq_counts):
2114     sorted_counts = sorted(freq_counts.values(), reverse=True)
2115     ranks = np.arange(1, len(sorted_counts) + 1)
2116     plt.figure(figsize=(10, 6))
2117     plt.loglog(ranks, sorted_counts, marker="o", color = "black", linestyle="none",
        label="Observed Data")
2118     plt.xlabel("Rank (log scale)")
2119     plt.ylabel("Frequency (log scale)")
2120     plt.title("Log-Log Plot of Rank-Frequency Distribution")
2121     plt.grid(True, which="both", linestyle="--", linewidth=0.5)
2122     plt.legend()
2123     plt.savefig("long-tail.png")
2124     plt.show()
2125

```



```

2126 # Fit Power-Law Distribution
2127 def fit_power_law(freq_counts):
2128     sorted_counts = np.array(sorted(freq_counts.values(), reverse=True))
2129     ranks = np.arange(1, len(sorted_counts) + 1)
2130
2131     def power_law(x, alpha, beta):
2132         return beta * x ** -alpha
2133
2134     params, _ = curve_fit(power_law, ranks, sorted_counts, maxfev=10000)
2135     fitted_alpha, fitted_beta = params
2136
2137     # Compute the predicted values
2138     predicted = power_law(ranks, fitted_alpha, fitted_beta)
2139
2140     # Plot the observed vs. fitted data
2141     plt.figure(figsize=(10, 6))
2142     plt.loglog(ranks, sorted_counts, marker="o", linestyle="none", color =
2143         "black", label="Observed Data")
2144     plt.loglog(ranks, predicted, label=f"Fitted Power-Law ( $\alpha$ ={fitted_alpha:.2f})",
2145         linestyle="--", color = "black")
2146     plt.xlabel("Rank (log scale)")
2147     plt.ylabel("Frequency (log scale)")
2148     plt.title("Power-Law Fit to Rank-Frequency Distribution")
2149     plt.grid(True, which="both", linestyle="--", linewidth=0.5)
2150     plt.savefig("power-law.png")
2151     plt.legend()
2152     plt.show()
2153
2154     mse = mean_squared_error(sorted_counts, predicted)
2155     print(f"Fitted Power-Law Parameters:  $\alpha$  = {fitted_alpha:.2f},  $\beta$  = {fitted_beta:.2f}")
2156     print(f"Mean Squared Error of Fit: {mse:.2f}")
2157
2158     return fitted_alpha, mse
2159
2160 # Calculate Gini Coefficient
2161 def calculate_gini(freq_counts):
2162     frequencies = np.array(sorted(freq_counts.values()))
2163     n = len(frequencies)
2164     cumulative_sum = np.cumsum(frequencies)
2165     gini = (n + 1 - 2 * np.sum(cumulative_sum) / cumulative_sum[-1]) / n
2166     print(f"Gini Coefficient: {gini:.2f}")
2167     return gini
2168
2169 # Example Usage
2170 # Assuming train_df["text_prepro"] contains the preprocessed text data
2171 freq_counts = tokenize_and_count(train_df["text_prepro"])
2172
2173 # Visualize Rank-Frequency Distribution
2174 plot_log_log_rank_frequency(freq_counts)
2175
2176 # Fit and Evaluate Power-Law Model
2177 fit_alpha, mse = fit_power_law(freq_counts)
2178
2179 # Calculate Gini Coefficient
2180 gini_coefficient = calculate_gini(freq_counts)
2181
2182 # Long-Tail Determination
2183 if fit_alpha > 1 and gini_coefficient > 0.5:
2184     print("The data displays long-tail behavior.")
2185 else:
2186     print("The data does not exhibit long-tail behavior.")
2187
2188 import pandas as pd
2189 import numpy as np
2190 import matplotlib.pyplot as plt
2191 from sklearn.feature_extraction.text import CountVectorizer
2192 from nltk.corpus import stopwords

```

```

2193 from nltk.stem import WordNetLemmatizer
2194 from wordcloud import WordCloud
2195
2196 # Step 1: Preprocess Text Data
2197 def preprocess_text(text):
2198     """Tokenize, remove stopwords, and lemmatize."""
2199     lemmatizer = WordNetLemmatizer()
2200     stop_words = set(stopwords.words("english"))
2201     tokens = text.lower().split() # Tokenize and lowercase
2202     tokens = [word for word in tokens if word.isalpha() and word not in stop_words]
2203     tokens = [lemmatizer.lemmatize(word) for word in tokens]
2204     return " ".join(tokens)
2205
2206 train_df["processed_text"] = train_df["text_prepro"].apply(preprocess_text)
2207
2208 # Step 2: Ensure datetime column exists
2209 train_df["at"] = pd.to_datetime(train_df["at"])
2210
2211 # Step 3: Group by time intervals (e.g., weekly)
2212 train_df["week"] = train_df["at"].dt.to_period("W")
2213 grouped_text = train_df.groupby("week")["processed_text"].apply(lambda x: " ".join(x))
2214
2215 # Step 4: Compute Term Frequencies
2216 vectorizer = CountVectorizer()
2217 term_matrix = vectorizer.fit_transform(grouped_text.values)
2218 terms = vectorizer.get_feature_names_out()
2219 term_frequencies = pd.DataFrame(term_matrix.toarray(), index=grouped_text.index,
2220                                columns=terms)
2221
2222 # Step 5: Normalize Frequencies (Relative Frequencies)
2223 normalized_frequencies = term_frequencies.div(term_frequencies.sum(axis=1), axis=0)
2224
2225 # Step 6: Identify Emerging Trends
2226 # Calculate the average rate of frequency increase
2227 frequency_trend = normalized_frequencies.diff().mean(axis=0).sort_values(ascending=False)
2228 top_emerging_terms = frequency_trend.head(10).index
2229
2230 # Step 7: Visualize Emerging Trends - Time Series Plot
2231 plt.figure(figsize=(12, 8))
2232 for term in top_emerging_terms:
2233     plt.plot(normalized_frequencies.index.to_timestamp(), normalized_frequencies[term],
2234             label=term, color = "black")
2235
2236 plt.title("Emerging Trends Over Time")
2237 plt.xlabel("Time (Weeks)")
2238 plt.ylabel("Relative Frequency")
2239 plt.legend(title="Terms")
2240 plt.grid()
2241 plt.savefig("emerging_trends.png")
2242 plt.show()
2243
2244 # Step 8: Visualize Emerging Trends - Word Cloud
2245 wordcloud = WordCloud(width=800, height=400,
2246                       background_color="white").generate_from_frequencies(
2247     frequency_trend.to_dict()
2248 )
2249 plt.figure(figsize=(10, 6))
2250 plt.imshow(wordcloud, interpolation="bilinear")
2251 plt.axis("off")
2252 plt.title("Emerging Terms Word Cloud")
2253 plt.show()
2254
2255 import numpy as np
2256 import pandas as pd
2257 from collections import Counter
2258 import matplotlib.pyplot as plt
2259

```

```

2259 # Tokenize and count token frequencies
2260 def get_token_frequencies(data):
2261     all_tokens = []
2262     for text in data:
2263         all_tokens.extend(text.split()) # Tokenize by splitting on spaces
2264     token_counts = Counter(all_tokens)
2265     return sorted(token_counts.items(), key=lambda x: x[1], reverse=True) # Sort by
        frequency
2266
2267 # Compute partition-specific contributions
2268 def partition_contributions(frequencies, num_partitions=10):
2269     total_tokens = sum(freq for token, freq in frequencies) # Total frequency
2270     partition_size = len(frequencies) // num_partitions # Number of tokens per partition
2271
2272     partition_results = {
2273         "Partition": [],
2274         "Cumulative Contribution (Q)": [],
2275         "Weighted Contribution (G)": [],
2276     }
2277
2278     for i in range(num_partitions):
2279         start_idx = i * partition_size
2280         end_idx = (i + 1) * partition_size if i != num_partitions - 1 else
            len(frequencies)
2281
2282         partition = frequencies[start_idx:end_idx]
2283         partition_cumulative = sum(freq for token, freq in partition)
2284         partition_weighted = sum(rank * freq for rank, (token, freq) in
            enumerate(partition, start=start_idx + 1))
2285
2286         partition_results["Partition"].append(f"Partition {i + 1}")
2287         partition_results["Cumulative Contribution (Q)"].append(partition_cumulative /
            total_tokens)
2288         partition_results["Weighted Contribution (G)"].append(partition_weighted /
            total_tokens)
2289
2290     return pd.DataFrame(partition_results)
2291
2292 # Visualize the contributions
2293 def visualize_partition_contributions(partition_df):
2294     x = np.arange(len(partition_df))
2295
2296     plt.figure(figsize=(12, 6))
2297     plt.bar(x - 0.2, partition_df["Cumulative Contribution (Q)"], width=0.4,
        label="Cumulative Contribution (Q)")
2298     plt.bar(x + 0.2, partition_df["Weighted Contribution (G)"], width=0.4,
        label="Weighted Contribution (G)")
2299     plt.xticks(x, partition_df["Partition"], rotation=45)
2300     plt.xlabel("Partitions")
2301     plt.ylabel("Contribution")
2302     plt.title("Partition-Specific Contributions (Q and G)")
2303     plt.legend()
2304     plt.tight_layout()
2305     plt.show()
2306
2307 # Main Workflow
2308 if __name__ == "__main__":
2309     # Load the data (replace with your dataset)
2310     token_frequencies = get_token_frequencies(train_df["text_prepro"]) # Extract token
        frequencies
2311
2312     # Partition-specific contributions
2313     partition_df = partition_contributions(token_frequencies, num_partitions=10)
2314
2315     # Visualize the results
2316     visualize_partition_contributions(partition_df)
2317
2318     # Display the DataFrame
2319     print(partition_df)

```

```

2320
2321
2322
2323 import numpy as np
2324 import matplotlib.pyplot as plt
2325 from collections import Counter
2326
2327 # Tokenize and compute token frequencies
2328 def compute_token_frequencies(data):
2329     all_tokens = " ".join(data).split()
2330     return Counter(all_tokens)
2331
2332 # Compute cumulative contribution Q(z)
2333 def compute_Q_function(frequencies, thresholds):
2334     sorted_freqs = np.array(sorted(frequencies.values(), reverse=True))
2335     Q_values = [sum(sorted_freqs[:z]) for z in thresholds]
2336     return Q_values
2337
2338 # Detect new peaks in Q(z)
2339 def detect_peaks(Q_values, thresholds, relative_increase=0.2):
2340     peaks = []
2341     for i in range(1, len(Q_values)):
2342         delta = Q_values[i] - Q_values[i-1]
2343         relative_change = delta / Q_values[i-1] if Q_values[i-1] > 0 else 0
2344         if relative_change > relative_increase: # Significant jump
2345             peaks.append((thresholds[i], Q_values[i]))
2346     return peaks
2347
2348 # Main execution
2349 data = train_df["text_prepro"] # Replace with your column
2350 token_frequencies = compute_token_frequencies(data)
2351
2352 # Define thresholds
2353 thresholds = range(1, len(token_frequencies) + 1, 10) # Every 10th token for efficiency
2354
2355 # Compute Q(z)
2356 Q_values = compute_Q_function(token_frequencies, thresholds)
2357
2358 # Detect peaks
2359 peaks = detect_peaks(Q_values, thresholds)
2360
2361 # Plot Q(z)
2362 plt.figure(figsize=(10, 6))
2363 plt.plot(thresholds, Q_values, '-o', label="Q(z)", markersize=4, color = 'black')
2364 plt.xlabel("Threshold (z)")
2365 plt.ylabel("Cumulative Contribution Q(z)")
2366 plt.title("Q-Function with Detected Peaks")
2367 plt.grid()
2368 for z, Q in peaks:
2369     plt.axvline(x=z, color='black', linestyle='--', alpha=0.7)
2370     plt.text(z, Q, f"Peak @ z={z}", rotation=90, color='black')
2371 plt.legend()
2372 plt.savefig("Q-Function with Detected Peaks.png")
2373 plt.show()
2374
2375 # Display peaks
2376 print("Detected Peaks (Threshold z, Q(z)):")
2377 for z, Q in peaks:
2378     print(f"Threshold z={z}, Q(z)={Q:.2f}")
2379
2380
2381
2382 from collections import Counter
2383 import pandas as pd
2384 import matplotlib.pyplot as plt
2385
2386 # Assuming token_frequencies is a Counter object (e.g., token_frequencies =
Counter(tokens))
2387 # Sort the tokens by frequency in descending order

```

```

2388 sorted_tokens = pd.DataFrame(token_frequencies.items(), columns=["Token", "Frequency"])
2389 sorted_tokens = sorted_tokens.sort_values(by="Frequency", ascending=False)
2390
2391 # Extract thresholds z=11 and z=21
2392 threshold_z11 = sorted_tokens[:11]
2393 threshold_z21 = sorted_tokens[:21]
2394
2395 # Display summaries
2396 print("Summary of Tokens at z=11:")
2397 print(threshold_z11)
2398
2399 print("\nSummary of Tokens at z=21:")
2400 print(threshold_z21)
2401
2402 # Plot contributions at z=11
2403 plt.figure(figsize=(10, 5))
2404 plt.bar(threshold_z11["Token"], threshold_z11["Frequency"], color='black')
2405 plt.title("Token Contributions at z=11")
2406 plt.ylabel("Frequency")
2407 plt.xlabel("Tokens")
2408 plt.xticks(rotation=45)
2409 plt.show()
2410
2411 # Plot contributions at z=21
2412 plt.figure(figsize=(10, 5))
2413 plt.bar(threshold_z21["Token"], threshold_z21["Frequency"], color='gray')
2414 plt.title("Token Contributions at z=21")
2415 plt.ylabel("Frequency")
2416 plt.xlabel("Tokens")
2417 plt.xticks(rotation=45)
2418 plt.show()
2419
2420
2421
2422 from textblob import TextBlob
2423 import pandas as pd
2424 import matplotlib.pyplot as plt
2425
2426 # Check if train_df and the 'text_prepro' column exist
2427 if "text_prepro" not in train_df.columns:
2428     raise ValueError("Column 'text_prepro' not found in train_df. Please check your
2429         dataset.")
2430
2431 # Ensure the column is not empty or full of null values
2432 if train_df["text_prepro"].isnull().all():
2433     raise ValueError("Column 'text_prepro' is empty. Please ensure it contains
2434         preprocessed reviews.")
2435
2436 # Function to analyze sentiment for a single review
2437 def analyze_sentiment(text):
2438     """
2439     Analyze sentiment of a given text using TextBlob.
2440     Returns 'Positive', 'Negative', or 'Neutral' based on polarity.
2441     """
2442     if not isinstance(text, str):
2443         return "Neutral" # Default for non-text entries
2444     polarity = TextBlob(text).sentiment.polarity
2445     if polarity > 0:
2446         return "Positive"
2447     elif polarity < 0:
2448         return "Negative"
2449     else:
2450         return "Neutral"
2451
2452 # Apply sentiment analysis to the entire dataset
2453 train_df["Sentiment"] = train_df["text_prepro"].apply(analyze_sentiment)
2454
2455 # Count the sentiment distribution
2456 sentiment_distribution = train_df["Sentiment"].value_counts()

```

```

2455
2456 # Print sentiment distribution
2457 print("Sentiment Distribution:\n", sentiment_distribution)
2458
2459 # Visualize sentiment distribution
2460 plt.figure(figsize=(10, 6))
2461 sentiment_distribution.plot(kind="bar", color=["black", "gray", "white"])
2462 plt.title("Sentiment Distribution Across Reviews")
2463 plt.xlabel("Sentiment")
2464 plt.ylabel("Number of Reviews")
2465 plt.xticks(rotation=0)
2466 plt.grid(axis="y", linestyle="--", linewidth=0.5)
2467 plt.tight_layout()
2468 plt.savefig("sentiment_distribution_large_dataset.png", dpi=300)
2469 plt.show()
2470
2471 # Display a few reviews for each sentiment category
2472 for sentiment_class in ["Positive", "Negative", "Neutral"]:
2473     print(f"\nSample {sentiment_class} Reviews:")
2474     sample_reviews = train_df[train_df["Sentiment"] ==
2475                               sentiment_class]["text_prepro"].head(5)
2476     print(sample_reviews)
2477
2478
2479 import pandas as pd
2480 import matplotlib.pyplot as plt
2481 import numpy as np
2482
2483 # Simulated Data: Replace with actual results
2484 data = {
2485     "Time Steps": [0, 1, 2, 3, 4, 5],
2486     "Q(z=10)": [10, 12, 15, 9, 6, 4],
2487     "Q(z=30)": [20, 25, 35, 50, 60, 75],
2488     "Q(z=50)": [30, 40, 60, 80, 100, 120],
2489 }
2490
2491 # Convert data into a DataFrame
2492 df = pd.DataFrame(data)
2493
2494 # Set Time Steps as the index for analysis
2495 df.set_index("Time Steps", inplace=True)
2496 print(df)
2497
2498 # Output:
2499 #           Q(z=10)  Q(z=30)  Q(z=50)
2500 # Time Steps
2501 # 0              10       20       30
2502 # 1              12       25       40
2503 # 2              15       35       60
2504 # 3               9       50       80
2505 # 4               6       60      100
2506 # 5               4       75      120
2507
2508
2509
2510 # Plotting Q(z) trends for each threshold
2511 plt.figure(figsize=(10, 6))
2512 for column in df.columns:
2513     plt.plot(df.index, df[column], marker='o', linestyle='-', label=column)
2514
2515 plt.title("Temporal Evolution of Q(z) Across Thresholds")
2516 plt.xlabel("Time Steps")
2517 plt.ylabel("Cumulative Contribution Q(z)")
2518 plt.legend(title="Thresholds")
2519 plt.grid()
2520 plt.show()
2521
2522

```

```

2523
2524 # Compute the rate of change for Q(z)
2525 rate_of_change = df.diff().dropna()
2526
2527 # Visualize rate of change
2528 plt.figure(figsize=(10, 6))
2529 for column in rate_of_change.columns:
2530     plt.plot(rate_of_change.index, rate_of_change[column], marker='o', color = 'black',
2531             linestyle='--', label=f"Rate of Change {column}")
2532
2533 plt.title("Rate of Change in Q(z) Over Time")
2534 plt.xlabel("Time Steps")
2535 plt.ylabel("Change in Q(z)")
2536 plt.legend(title="Thresholds")
2537 plt.grid()
2538 plt.show()
2539
2540 print("Rate of Change Table:")
2541 print(rate_of_change)
2542
2543
2544 # Persistence Analysis: Summarize final cumulative contribution for each threshold
2545 final_contribution = df.iloc[-1]
2546
2547 # Plot bar chart for persistence
2548 plt.figure(figsize=(8, 5))
2549 plt.bar(final_contribution.index, final_contribution.values, color='gray')
2550 plt.title("Final Cumulative Contributions at Different Thresholds")
2551 plt.xlabel("Thresholds (z)")
2552 plt.ylabel("Q(z) at Final Time Step")
2553 plt.savefig("Cumulative Contributions at Different Thresholds.png")
2554 plt.show()
2555
2556 print("Final Contributions:")
2557 print(final_contribution)
2558
2559
2560
2561 import plotly.express as px
2562
2563 # Melt the DataFrame for long-format plotting
2564 long_df = df.reset_index().melt(id_vars="Time Steps", var_name="Threshold",
2565                                value_name="Q(z)")
2566
2567 # Create an interactive line plot
2568 fig = px.line(long_df, x="Time Steps", y="Q(z)", color="Threshold", markers=True,
2569              title="Interactive Temporal Evolution of Q(z)",
2570              labels={"Q(z)": "Cumulative Contribution Q(z)"})
2571
2572 fig.show()
2573
2574
2575 # Re-import necessary libraries after execution state reset
2576 import pandas as pd
2577 import numpy as np
2578 import matplotlib.pyplot as plt
2579 from sklearn.cluster import KMeans
2580 from scipy.signal import find_peaks
2581
2582 # Simulated Q(z) data (replace with actual values or load precomputed data)
2583 thresholds = np.arange(1, 101)
2584 Q_z = np.cumsum(np.random.randint(1, 20, size=len(thresholds))) # Simulated cumulative
2585 Q(z)
2586
2587 # Find the peaks in Q(z) for optimal thresholds
2588 peaks, _ = find_peaks(Q_z, prominence=10)

```

```

2589 # Plot Q(z) with detected peaks
2590 plt.figure(figsize=(10, 6))
2591 plt.plot(thresholds, Q_z, label="Cumulative Q(z)", color='black')
2592 plt.scatter(thresholds[peaks], Q_z[peaks], color='red', zorder=5, label="Detected Peaks")
2593 plt.title("Threshold Optimization for Q(z)")
2594 plt.xlabel("Threshold (z)")
2595 plt.ylabel("Cumulative Contribution Q(z)")
2596 plt.legend()
2597 plt.grid()
2598 plt.show()
2599
2600 # Highlight thresholds of interest
2601 optimal_thresholds = thresholds[peaks]
2602 optimal_Q_values = Q_z[peaks]
2603
2604 threshold_opt_df = pd.DataFrame({
2605     "Threshold": optimal_thresholds,
2606     "Q(z)": optimal_Q_values
2607 })
2608
2609 threshold_opt_df
2610
2611
2612
2613 from collections import Counter
2614 from sklearn.preprocessing import StandardScaler
2615
2616 # Simulated token-frequency data for higher thresholds (replace with actual data)
2617 token_frequencies = Counter({
2618     "rare_token_1": 50, "rare_token_2": 60, "common_token_1": 1000,
2619     "common_token_2": 950, "niche_token_1": 70, "niche_token_2": 80
2620 })
2621
2622 # Convert token frequencies to DataFrame
2623 token_df = pd.DataFrame(token_frequencies.items(), columns=["Token", "Frequency"])
2624
2625 # Preprocess data for clustering
2626 X = token_df["Frequency"].values.reshape(-1, 1)
2627 scaler = StandardScaler()
2628 X_scaled = scaler.fit_transform(X)
2629
2630 # Apply K-Means clustering
2631 kmeans = KMeans(n_clusters=2, random_state=42)
2632 token_df["Cluster"] = kmeans.fit_predict(X_scaled)
2633
2634 # Visualize clusters
2635 plt.figure(figsize=(10, 6))
2636 for cluster in token_df["Cluster"].unique():
2637     cluster_data = token_df[token_df["Cluster"] == cluster]
2638     plt.bar(cluster_data["Token"], cluster_data["Frequency"], label=f"Cluster {cluster}")
2639 plt.xticks(rotation=45)
2640 plt.title("Cluster Analysis of Token Contributions")
2641 plt.xlabel("Tokens")
2642 plt.ylabel("Frequency")
2643 plt.legend()
2644 plt.show()
2645
2646 token_df
2647
2648
2649
2650

```