```
Start coding or generate with AI.
```

```
print("Installing nltk...")
!pip install nltk
print("nltk installed.")
```

```
Installing nltk...
Requirement already satisfied: nltk in /usr/local/lib/python3.12/dist-packages (3.9.1)
Requirement already satisfied: click in /usr/local/lib/python3.12/dist-packages (from nltk) (8.3.1
Requirement already satisfied: joblib in /usr/local/lib/python3.12/dist-packages (from nltk) (1.5.
Requirement already satisfied: regex>=2021.8.3 in /usr/local/lib/python3.12/dist-packages (from nl
Requirement already satisfied: tqdm in /usr/local/lib/python3.12/dist-packages (from nltk) (4.67.1
nltk installed.
```

```
print("Installing numpy and pandas...")
!pip install numpy pandas
print("numpy and pandas installed.")
```

```
Installing numpy and pandas...
Requirement already satisfied: numpy in /usr/local/lib/python3.12/dist-packages (2.0.2)
Requirement already satisfied: pandas in /usr/local/lib/python3.12/dist-packages (2.2.2)
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.12/dist-packages (
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.12/dist-packages (from panda
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.12/dist-packages (from pan
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.12/dist-packages (from python-da
numpy and pandas installed.
```

```
print("Installing matplotlib...")
!pip install matplotlib
print("matplotlib installed.")
```

```
Installing matplotlib...
Requirement already satisfied: matplotlib in /usr/local/lib/python3.12/dist-packages (3.10.0)
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.12/dist-packages (from m
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.12/dist-packages (from matpl
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.12/dist-packages (from
Requirement already satisfied: kiwisolver>=1.3.1 in /usr/local/lib/python3.12/dist-packages (from
Requirement already satisfied: numpy>=1.23 in /usr/local/lib/python3.12/dist-packages (from matplo
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.12/dist-packages (from ma
Requirement already satisfied: pillow>=8 in /usr/local/lib/python3.12/dist-packages (from matplotl
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.12/dist-packages (from m
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/python3.12/dist-packages (fr
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.12/dist-packages (from python-da
matplotlib installed.
```

```
import nltk

# Ensure 'punkt' tokenizer data is downloaded
try:
    nltk.data.find('tokenizers/punkt')
    print("'punkt' tokenizer data is already downloaded.")
except LookupError:
    print("Downloading 'punkt' tokenizer data...")
    nltk.download('punkt')
    print("'punkt' tokenizer data downloaded.")

# Ensure 'punkt_tab' tokenizer data is downloaded (often needed for word_tokenize)
```

```
try:
    nltk.data.find('tokenizers/punkt_tab')
    print("'punkt_tab' tokenizer data is already downloaded.")
except LookupError:
    print("Downloading 'punkt_tab' tokenizer data...")
    nltk.download('punkt_tab')
    print("'punkt_tab' tokenizer data downloaded.")
```

```
'punkt' tokenizer data is already downloaded.
Downloading 'punkt_tab' tokenizer data...
'punkt_tab' tokenizer data downloaded.
[nltk_data] Downloading package punkt_tab to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt_tab.zip.
```

```
text = """
Natural language processing is a branch of artificial intelligence that focuses on the interactic
 It enables machines to understand interpret and generate human language in a meaningful way.
 Language models play a crucial role in many natural language processing applications.
  A language model predicts the probability of a sequence of words occurring in a sentence"""*40
```

Text Preprocessing Steps

```
import nltk
import re
from nltk.corpus import stopwords

# Ensure stopwords are downloaded
try:
    nltk.data.find('corpora/stopwords')
    print("Stopwords corpus is already downloaded.")
except LookupError:
    print("Downloading stopwords corpus...")
    nltk.download('stopwords')
    print("Stopwords corpus downloaded.")
```

```
Downloading stopwords corpus...
Stopwords corpus downloaded.
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
```

Step 1: Convert to lowercase and remove punctuation/numbers

```
text = """
Natural language processing is a branch of artificial intelligence that focuses on the interactic
 It enables machines to understand interpret and generate human language in a meaningful way.
 Language models play a crucial role in many natural language processing applications.
  A language model predicts the probability of a sequence of words occurring in a sentence"""*40

# Step 1: Convert to lowercase
lowercased_text = text.lower()

print("Text after lowercasing (first 500 chars):")
print(lowercased_text[:500])
```

```
Text after lowercasing (first 500 chars):

natural language processing is a branch of artificial intelligence that focuses on the interaction
 it enables machines to understand interpret and generate human language in a meaningful way.
```

```
 language models play a crucial role in many natural language processing applications.
  a language model predicts the probability of a sequence of words occurring in a sentence
natural language processing is a branch of artificial intelligence that focuses on the int
```

Step 2: Tokenize words

```
from nltk.tokenize import word_tokenize, sent_tokenize
import re # Ensure re is imported for this step too

# Step 2: Tokenize into sentences (punctuation still present for accurate splitting)
sentences = sent_tokenize(lowercased_text)

# Now, for each sentence, perform word-level cleaning and tokenization
processed_sentences_words = []
for sentence in sentences:
    # Remove punctuation and numbers from the current sentence
    cleaned_sentence = re.sub(r'[^a-zA-Z\s]', '', sentence)
    # Tokenize words
    words = word_tokenize(cleaned_sentence)
    processed_sentences_words.append(words)

# Flatten the list of lists for a total word count, if desired for overall stats
all_word_tokens_flat = [word for sublist in processed_sentences_words for word in sublist]

print(f"Total sentences tokenized: {len(processed_sentences_words)}")
print("First 2 sentences (word tokens after cleaning) :")
for i, s_words in enumerate(processed_sentences_words[:2]):
    print(f"Sentence {i+1}: {s_words[:20]}...") # show first 20 words of each sentence
print(f"Total words after initial cleaning and tokenization: {len(all_word_tokens_flat)}")

# We will now use 'processed_sentences_words' (list of lists of words) for subsequent steps
```

```
Total sentences tokenized: 121
First 2 sentences (word tokens after cleaning) :
Sentence 1: ['natural', 'language', 'processing', 'is', 'a', 'branch', 'of', 'artificial', 'intell
Sentence 2: ['it', 'enables', 'machines', 'to', 'understand', 'interpret', 'and', 'generate', 'hum
Total words after initial cleaning and tokenization: 2400
```

Step 3 (Optional): Remove stopwords

```
from nltk.corpus import stopwords # Ensure stopwords are imported

stop_words = set(stopwords.words('english'))

# Step 3 (Optional): Remove stopwords from each sentence's word list
filtered_sentences_words = []
for sentence_word_list in processed_sentences_words:
    filtered_words = [word for word in sentence_word_list if word not in stop_words]
    filtered_sentences_words.append(filtered_words)

# Flatten the list of lists for a total word count after stopword removal
all_filtered_word_tokens_flat = [word for sublist in filtered_sentences_words for word in sublist

print(f"Total sentences after stopword removal: {len(filtered_sentences_words)}")
print("First 2 sentences (word tokens after stopword removal) :")
for i, s_words in enumerate(filtered_sentences_words[:2]):
    print(f"Sentence {i+1}: {s_words[:20]}...") # show first 20 words of each sentence
```

```
print(f"Total words after stopword removal: {len(all_filtered_word_tokens_flat)}")

# 'filtered_sentences_words' is now a list of lists, where each inner list contains words of a se
```

```
Total sentences after stopword removal: 121
First 2 sentences (word tokens after stopword removal) :
Sentence 1: ['natural', 'language', 'processing', 'branch', 'artificial', 'intelligence', 'focuses
Sentence 2: ['enables', 'machines', 'understand', 'interpret', 'generate', 'human', 'language', 'm
Total words after stopword removal: 1520
```

Step 4: Add start/end tokens for sentences

```
# Step 4: Add start and end tokens to each sentence
sentences_with_tokens = []
for sentence_word_list in filtered_sentences_words:
    tokenized_sentence = ['<START>'] + sentence_word_list + ['<END>']
    sentences_with_tokens.append(tokenized_sentence)

print(f"Total sentences with start/end tokens: {len(sentences_with_tokens)}")
print("First 2 sentences with start/end tokens:")
for i, s in enumerate(sentences_with_tokens[:2]):
    print(f"Sentence {i+1}: {s[:20]}...<END>") # Displaying first 20 words and <END> token

# 'sentences_with_tokens' now contains your fully preprocessed data.
# Each element is a list of words for a sentence, including <START> and <END> tokens.
```

```
Total sentences with start/end tokens: 121
First 2 sentences with start/end tokens:
Sentence 1: ['<START>', 'natural', 'language', 'processing', 'branch', 'artificial', 'intelligence
Sentence 2: ['<START>', 'enables', 'machines', 'understand', 'interpret', 'generate', 'human', 'la
```

```
Start coding or generate with AI.
```

Constructing Unigrams, Bigrams, and Trigrams

```
from nltk.util import ngrams
from collections import Counter

# --- Unigrams ---
all_unigrams = []
for sentence_tokens in sentences_with_tokens:
    # Unigrams are simply the words in each sentence
    all_unigrams.extend(sentence_tokens)

# Count frequency of unigrams
unigram_counts = Counter(all_unigrams)

print(f"Total Unigrams: {len(all_unigrams)}")
print("Most Common 10 Unigrams:")
for unigram, count in unigram_counts.most_common(10):
    print(f"  '{unigram}': {count}")
```

```
Total Unigrams: 1762
Most Common 10 Unigrams:
  'language': 240
  '<START>': 121
  '<END>': 121
```

```
  'natural': 80
  'processing': 80
  'human': 80
  'branch': 40
  'artificial': 40
  'intelligence': 40
  'focuses': 40
```

```
# --- Bigrams ---
all_bigrams = []
for sentence_tokens in sentences_with_tokens:
    # Generate bigrams for the current sentence
    bigrams_in_sentence = list(ngrams(sentence_tokens, 2))
    all_bigrams.extend(bigrams_in_sentence)

# Count frequency of bigrams
bigram_counts = Counter(all_bigrams)

print(f"Total Bigrams: {len(all_bigrams)}")
print("Most Common 10 Bigrams:")
for bigram, count in bigram_counts.most_common(10):
    print(f"  '{' '.join(bigram)}': {count}")
```

```
Total Bigrams: 1641
Most Common 10 Bigrams:
  'natural language': 80
  'language processing': 80
  'human language': 80
  '<START> language': 80
  'processing branch': 40
  'branch artificial': 40
  'artificial intelligence': 40
  'intelligence focuses': 40
  'focuses interaction': 40
  'interaction computers': 40
```

```
# --- Trigrams ---
all_trigrams = []
for sentence_tokens in sentences_with_tokens:
    # Generate trigrams for the current sentence
    trigrams_in_sentence = list(ngrams(sentence_tokens, 3))
    all_trigrams.extend(trigrams_in_sentence)

# Count frequency of trigrams
trigram_counts = Counter(all_trigrams)

print(f"Total Trigrams: {len(all_trigrams)}")
print("Most Common 10 Trigrams:")
for trigram, count in trigram_counts.most_common(10):
    print(f"  '{' '.join(trigram)}': {count}")
```

```
Total Trigrams: 1520
Most Common 10 Trigrams:
  'natural language processing': 80
  'language processing branch': 40
  'processing branch artificial': 40
  'branch artificial intelligence': 40
  'artificial intelligence focuses': 40
  'intelligence focuses interaction': 40
  'focuses interaction computers': 40
  'interaction computers human': 40
  'computers human language': 40
```

```
    'human language <END>': 40
```

```
Start coding or generate with AI.
```

Unigram Word Counts

```
import pandas as pd

# Convert unigram_counts to a pandas DataFrame
unigram_df = pd.DataFrame(unigram_counts.most_common(20), columns=['Unigram', 'Count'])

print("Top 20 Most Frequent Unigrams:")
display(unigram_df)
```

```
Top 20 Most Frequent Unigrams:
```

	Unigram	Count
0	language	240
1	<START>	121
2	<END>	121
3	natural	80
4	processing	80
5	human	80
6	branch	40
7	artificial	40
8	intelligence	40
9	focuses	40
10	interaction	40
11	computers	40
12	enables	40
13	machines	40
14	understand	40
15	interpret	40
16	generate	40
17	meaningful	40
18	way	40
19	models	40

Next steps: Generate code with unigram_df | New interactive sheet

Bigram Word Counts

```
# Convert bigram_counts to a pandas DataFrame
bigram_df = pd.DataFrame([(f"{word1} {word2}", count) for (word1, word2), count in bigram_counts.

print("Top 20 Most Frequent Bigrams:")
display(bigram_df)
```

```
Top 20 Most Frequent Bigrams:
```

	Bigram	Count
0	natural language	80
1	language processing	80
2	human language	80
3	<START> language	80
4	processing branch	40
5	branch artificial	40
6	artificial intelligence	40
7	intelligence focuses	40
8	focuses interaction	40
9	interaction computers	40
10	computers human	40
11	language <END>	40
12	<START> enables	40
13	enables machines	40
14	machines understand	40
15	understand interpret	40
16	interpret generate	40
17	generate human	40
18	language meaningful	40
19	meaningful way	40

Next steps: Generate code with `bigram_df` New interactive sheet

Trigram Word Counts

```
# Convert trigram_counts to a pandas DataFrame
trigram_df = pd.DataFrame([(f"{word1} {word2} {word3}", count) for (word1, word2, word3), count i

print("Top 20 Most Frequent Trigrams:")
display(trigram_df)
```

```
Top 20 Most Frequent Trigrams:
```

	Trigram	Count
0	natural language processing	80
1	language processing branch	40
2	processing branch artificial	40
3	branch artificial intelligence	40
4	artificial intelligence focuses	40
5	intelligence focuses interaction	40
6	focuses interaction computers	40
7	interaction computers human	40
8	computers human language	40
9	human language <END>	40
10	<START> enables machines	40
11	enables machines understand	40
12	machines understand interpret	40
13	understand interpret generate	40
14	interpret generate human	40
15	generate human language	40
16	human language meaningful	40
17	language meaningful way	40
18	meaningful way <END>	40
19	<START> language models	40

Next steps: Generate code with `trigram_df` New interactive sheet

```
Start coding or generate with AI.
```

Add-one (Laplace) Smoothing

```
import pandas as pd

# Calculate Vocabulary Size (V) - number of unique words including <START> and <END>
V = len(unigram_counts)

print(f"Vocabulary Size (V): {V}")
```

```
Vocabulary Size (V): 32
```

Unigram Smoothing

```
# Total number of words in the corpus (N)
N = len(all_unigrams)
```

```
# Calculate smoothed probabilities for unigrams
smoothed_unigram_data = []
for unigram, count in unigram_counts.items():
    smoothed_count = count + 1
    smoothed_probability = smoothed_count / (N + V)
    smoothed_unigram_data.append({
        'Unigram': unigram,
        'Raw Count': count,
        'Smoothed Count': smoothed_count,
        'Smoothed Probability': smoothed_probability
    })

smoothed_unigram_df = pd.DataFrame(smoothed_unigram_data)
smoothed_unigram_df = smoothed_unigram_df.sort_values(by='Raw Count', ascending=False).reset_inde

print("Top 20 Most Frequent Unigrams with Smoothed Probabilities:")
display(smoothed_unigram_df.head(20))
```

```
Top 20 Most Frequent Unigrams with Smoothed Probabilities:
```

0	language	240	241	0.134337
1	<START>	121	122	0.068004
2	<END>	121	122	0.068004
3	natural	80	81	0.045151
4	human	80	81	0.045151
5	processing	80	81	0.045151
6	branch	40	41	0.022854
7	artificial	40	41	0.022854
8	focuses	40	41	0.022854
9	intelligence	40	41	0.022854
10	computers	40	41	0.022854
11	interaction	40	41	0.022854
12	enables	40	41	0.022854
13	machines	40	41	0.022854
14	understand	40	41	0.022854
15	interpret	40	41	0.022854
16	generate	40	41	0.022854
17	meaningful	40	41	0.022854
18	way	40	41	0.022854
19	models	40	41	0.022854

Bigram Smoothing

```
# Calculate smoothed probabilities for bigrams
smoothed_bigram_data = []
for bigram, count in bigram_counts.items():
    w_prev_count = unigram_counts.get(bigram[0], 0) # Get count of the first word in the bigram
    smoothed_count = count + 1
    smoothed_probability = smoothed_count / (w_prev_count + V)
    smoothed_bigram_data.append({
        'Bigram': ' '.join(bigram),
        'Raw Count': count,
        'Context Count (C(w_prev))': w_prev_count,
        'Smoothed Count': smoothed_count,
        'Smoothed Probability': smoothed_probability
    })

smoothed_bigram_df = pd.DataFrame(smoothed_bigram_data)
smoothed_bigram_df = smoothed_bigram_df.sort_values(by='Raw Count', ascending=False).reset_index(

print("Top 20 Most Frequent Bigrams with Smoothed Probabilities:")
display(smoothed_bigram_df.head(20))
```

```
Top 20 Most Frequent Bigrams with Smoothed Probabilities:
```

	Bigram	Raw Count	Context Count (C(w_prev))	Smoothed Count	Smoothed Probability
0	natural language	80	80	81	0.723214
1	language processing	80	240	81	0.297794
2	human language	80	80	81	0.723214
3	<START> language	80	121	81	0.529412
4	branch artificial	40	40	41	0.569444
5	artificial intelligence	40	40	41	0.569444
6	processing branch	40	80	41	0.366071
7	models play	40	40	41	0.569444
8	intelligence focuses	40	40	41	0.569444
9	interaction computers	40	40	41	0.569444
10	focuses interaction	40	40	41	0.569444
11	language <END>	40	240	41	0.150735
12	<START> enables	40	121	41	0.267974
13	enables machines	40	40	41	0.569444
14	computers human	40	40	41	0.569444
15	machines understand	40	40	41	0.569444
16	understand interpret	40	40	41	0.569444
17	generate human	40	40	41	0.569444
18	interpret generate	40	40	41	0.569444

Trigram Smoothing

```
# Calculate smoothed probabilities for trigrams
smoothed_trigram_data = []
for trigram, count in trigram_counts.items():
    bigram_context = (trigram[0], trigram[1])
    w_prev_prev_prev_count = bigram_counts.get(bigram_context, 0) # Get count of the context bigr
    smoothed_count = count + 1
    smoothed_probability = smoothed_count / (w_prev_prev_prev_count + V)
    smoothed_trigram_data.append({
        'Trigram': ' '.join(trigram),
        'Raw Count': count,
        'Context Count (C(w_prev, w_prev_prev))': w_prev_prev_prev_count,
        'Smoothed Count': smoothed_count,
        'Smoothed Probability': smoothed_probability
    })

smoothed_trigram_df = pd.DataFrame(smoothed_trigram_data)
smoothed_trigram_df = smoothed_trigram_df.sort_values(by='Raw Count', ascending=False).reset_inde

print("Top 20 Most Frequent Trigrams with Smoothed Probabilities:")
display(smoothed_trigram_df.head(20))
```

Top 20 Most Frequent Trigrams with Smoothed Probabilities:

	Trigram	Raw Count	Context Count (C(w_prev, w_prev_prev))	Smoothed Count	Smoothed Probability
0	natural language processing	80	80	81	0.723214
1	language processing branch	40	80	41	0.366071
2	processing branch artificial	40	40	41	0.569444
3	artificial intelligence focuses	40	40	41	0.569444
4	branch artificial intelligence	40	40	41	0.569444
5	intelligence focuses interaction	40	40	41	0.569444
6	focuses interaction computers	40	40	41	0.569444
7	play crucial role	40	40	41	0.569444
8	interaction computers human	40	40	41	0.569444
9	computers human language	40	40	41	0.569444
10	human language <END>	40	80	41	0.366071
11	<START> enables machines	40	40	41	0.569444
12	enables machines understand	40	40	41	0.569444
13	machines understand interpret	40	40	41	0.569444
14	understand interpret generate	40	40	41	0.569444
15	interpret generate human	40	40	41	0.569444
16	generate human language	40	40	41	0.569444
17	human language meaningful	40	80	41	0.366071
18	language meaningful way	40	40	41	0.569444
19	meaningful way <END>	40	40	41	0.569444

```
Start coding or generate with AI.
```

```
test_sentences = [
    "natural language processing models predict probability of words",
    "artificial intelligence enables machines understand human language",
```

```
    "language models play crucial role in many applications",
    "a language model predicts a sequence of words in a sentence",
    "human language interaction focuses on computers and language models"
]

print(f"Defined {len(test_sentences)} test sentences.")
for i, sentence in enumerate(test_sentences):
    print(f"Sentence {i+1}: {sentence}")
```

```
Defined 5 test sentences.
Sentence 1: natural language processing models predict probability of words
Sentence 2: artificial intelligence enables machines understand human language
Sentence 3: language models play crucial role in many applications
Sentence 4: a language model predicts a sequence of words in a sentence
Sentence 5: human language interaction focuses on computers and language models
```

```
import numpy as np

def calculate_unigram_log_probability(sentence_tokens, unigram_counts, V, N):
    """Calculates the unigram log probability of a sentence using add-one smoothing."""
    log_probability = 0.0
    for word in sentence_tokens:
        # Get smoothed count for the word
        smoothed_count = unigram_counts.get(word, 0) + 1
        # Calculate smoothed probability
        smoothed_prob = smoothed_count / (N + V)
        log_probability += np.log(smoothed_prob)
    return log_probability

# Calculate unigram log probabilities for all preprocessed test sentences
unigram_sentence_log_probabilities = []
for i, sentence_tokens in enumerate(preprocessed_test_sentences):
    log_prob = calculate_unigram_log_probability(sentence_tokens, unigram_counts, V, N)
    unigram_sentence_log_probabilities.append(log_prob)
    print(f"Sentence {i+1} Unigram Log Probability: {log_prob:.4f}")
```

```
Sentence 1 Unigram Log Probability: -32.4074
Sentence 2 Unigram Log Probability: -29.3747
Sentence 3 Unigram Log Probability: -30.0556
Sentence 4 Unigram Log Probability: -26.2769
Sentence 5 Unigram Log Probability: -27.6035
```

```
def calculate_bigram_log_probability(sentence_tokens, bigram_counts, unigram_counts, V):
    """Calculates the bigram log probability of a sentence using add-one smoothing."""
    log_probability = 0.0
    # Iterate through bigrams in the sentence (starting from the second word)
    for i in range(len(sentence_tokens) - 1):
        w_prev = sentence_tokens[i]
        w_curr = sentence_tokens[i+1]
        bigram = (w_prev, w_curr)

        # Get smoothed count for the bigram C(w_prev, w_curr) + 1
        smoothed_bigram_count = bigram_counts.get(bigram, 0) + 1

        # Get count for the context C(w_prev) + V
        # If w_prev is not in unigram_counts (shouldn't happen with <START> token usually), it def
        # but V is added to ensure non-zero denominator.
        context_count = unigram_counts.get(w_prev, 0) + V

        smoothed_conditional_prob = smoothed_bigram_count / context_count
        log_probability += np.log(smoothed_conditional_prob)
```

```
        return log_probability

    # Calculate bigram log probabilities for all preprocessed test sentences
    bigram_sentence_log_probabilities = []
    for i, sentence_tokens in enumerate(preprocessed_test_sentences):
        log_prob = calculate_bigram_log_probability(sentence_tokens, bigram_counts, unigram_counts, V)
        bigram_sentence_log_probabilities.append(log_prob)
        print(f"Sentence {i+1} Bigram Log Probability: {log_prob:.4f}")
```

```
Sentence 1 Bigram Log Probability: -26.8869
Sentence 2 Bigram Log Probability: -17.4893
Sentence 3 Bigram Log Probability: -9.6204
Sentence 4 Bigram Log Probability: -15.7913
Sentence 5 Bigram Log Probability: -29.9592
```

Reasoning: After calculating unigram and bigram log probabilities, the next logical step is to calculate the trigram log probabilities for the test sentences. This will involve creating a function that applies add-one smoothing to trigram counts, using bigram counts as the context for the conditional probability, all on a logarithmic scale.

```
def calculate_trigram_log_probability(sentence_tokens, trigram_counts, bigram_counts, V):
    """Calculates the trigram log probability of a sentence using add-one smoothing."""
    log_probability = 0.0
    # Iterate through trigrams in the sentence (starting from the third word)
    for i in range(len(sentence_tokens) - 2):
        w_prev_prev = sentence_tokens[i]
        w_prev = sentence_tokens[i+1]
        w_curr = sentence_tokens[i+2]
        trigram = (w_prev_prev, w_prev, w_curr)
        bigram_context = (w_prev_prev, w_prev)

        # Get smoothed count for the trigram C(w_prev_prev, w_prev, w_curr) + 1
        smoothed_trigram_count = trigram_counts.get(trigram, 0) + 1

        # Get count for the context C(w_prev_prev, w_prev) + V
        context_count = bigram_counts.get(bigram_context, 0) + V

        smoothed_conditional_prob = smoothed_trigram_count / context_count
        log_probability += np.log(smoothed_conditional_prob)
    return log_probability

# Calculate trigram log probabilities for all preprocessed test sentences
trigram_sentence_log_probabilities = []
for i, sentence_tokens in enumerate(preprocessed_test_sentences):
    log_prob = calculate_trigram_log_probability(sentence_tokens, trigram_counts, bigram_counts, V
    trigram_sentence_log_probabilities.append(log_prob)
    print(f"Sentence {i+1} Trigram Log Probability: {log_prob:.4f}")
```

```
Sentence 1 Trigram Log Probability: -21.7089
Sentence 2 Trigram Log Probability: -20.5186
Sentence 3 Trigram Log Probability: -10.9997
Sentence 4 Trigram Log Probability: -17.0528
Sentence 5 Trigram Log Probability: -26.3238
```

Perplexity Calculation

```
import numpy as np
```

```
def calculate_perplexity(log_probability, num_words):
    """Calculates perplexity from log probability and number of words."""
    # Perplexity = e ^ ( - (log_probability / num_words) )
    return np.exp(-log_probability / num_words)

# Calculate Perplexity for Unigram Model
unigram_perplexities = []
for i, log_prob in enumerate(unigram_sentence_log_probabilities):
    # For unigram, num_words includes <START> and <END> tokens for consistency in calculation
    num_words = len(preprocessed_test_sentences[i])
    perplexity = calculate_perplexity(log_prob, num_words)
    unigram_perplexities.append(perplexity)

# Calculate Perplexity for Bigram Model
bigram_perplexities = []
for i, log_prob in enumerate(bigram_sentence_log_probabilities):
    # For bigram, num_words is (length - 1) because the first word has no preceding word
    num_words = len(preprocessed_test_sentences[i]) - 1
    if num_words == 0: # Handle cases where sentence might be too short for bigram
        bigram_perplexities.append(np.inf) # Assign infinity if no bigrams possible
    else:
        perplexity = calculate_perplexity(log_prob, num_words)
        bigram_perplexities.append(perplexity)

# Calculate Perplexity for Trigram Model
trigram_perplexities = []
for i, log_prob in enumerate(trigram_sentence_log_probabilities):
    # For trigram, num_words is (length - 2) because the first two words have no preceding trigr
    num_words = len(preprocessed_test_sentences[i]) - 2
    if num_words <= 0: # Handle cases where sentence might be too short for trigram
        trigram_perplexities.append(np.inf) # Assign infinity if no trigrams possible
    else:
        perplexity = calculate_perplexity(log_prob, num_words)
        trigram_perplexities.append(perplexity)

# Add perplexity to the DataFrame
probabilities_df['Unigram Perplexity'] = unigram_perplexities
probabilities_df['Bigram Perplexity'] = bigram_perplexities
probabilities_df['Trigram Perplexity'] = trigram_perplexities

print("Summary of Sentence Log Probabilities and Perplexities:")
display(probabilities_df)
```

```
Summary of Sentence Log Probabilities and Perplexities:
```

	Sentence	Unigram Log Probability	Bigram Log Probability	Trigram Log Probability	Unigram Perplexity	Bigram Perplexity	Trigram Perplexity
0	natural language processing	-32.407374	-26.886927	-21.708853	36.628232	28.814128	22.226042