```
import nltk
from collections import Counter
import math

# Download necessary NLTK data for tokenization and other text processing tasks
# 'punkt' is a pre-trained tokenizer for English.
# 'stopwords' provides a list of common words to be filtered out (optional, but g
nltk.download('punkt')
nltk.download('stopwords')
nltk.download('punkt_tab') # Added to download the missing resource for sent_toke
```

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package punkt_tab to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt_tab.zip.
True
```

```
# Define a sample text corpus (replace with loading from a file if needed)
raw_corpus = """
This is the first sentence of our sample text. It discusses natural language proc


N-grams are sequences of N words.

Another paragraph follows. This one is about the importance of clean data.
Text processing is crucial for accurate analysis.

End of sample.
"""

print("--- Raw Corpus (first 100 chars) ---")
print(raw_corpus[:100] + '...' if len(raw_corpus) > 100 else raw_corpus)
print("\n--- Raw Corpus Lines Count ---")
print(f"Total lines in raw corpus: {len(raw_corpus.splitlines())}")
```

```
--- Raw Corpus (first 100 chars) ---

This is the first sentence of our sample text. It discusses natural language pro


N-grams ...

--- Raw Corpus Lines Count ---
Total lines in raw corpus: 10
```

```
# Clean unnecessary lines (e.g., empty lines, or lines with only whitespace)
# We'll split the corpus into lines, remove leading/trailing whitespace, and fi
cleaned_lines = []
for line in raw_corpus.splitlines():
    stripped_line = line.strip()
    if stripped_line: # Only add non-empty lines
        cleaned_lines.append(stripped_line)
```

```
# Join the cleaned lines back into a single string or keep as a list of lines
cleaned_corpus = "\n".join(cleaned_lines)

print("--- Cleaned Corpus Sample (first 10 lines) ---")
# Displaying the first few lines of the cleaned corpus
for i, line in enumerate(cleaned_corpus.splitlines()):
    if i < 10: # Display up to 10 lines
        print(line)
    else:
        print("...")
        break

print(f"\nTotal lines in cleaned corpus: {len(cleaned_corpus.splitlines())}")
```

```
--- Cleaned Corpus Sample (first 10 lines) ---
This is the first sentence of our sample text. It discusses natural language pro
N-grams are sequences of N words.
Another paragraph follows. This one is about the importance of clean data.
Text processing is crucial for accurate analysis.
End of sample.

Total lines in cleaned corpus: 5
```

```
def to_lowercase(text):
    """Converts all characters in a string to lowercase."""
    return text.lower()
```

```
import re
import string

def remove_punctuation_and_numbers(text):
    """Removes punctuation and numbers from a string."""
    # Remove numbers
    text = re.sub(r'\d+', '', text)
    # Remove punctuation
    text = text.translate(str.maketrans('', '', string.punctuation))
    return text
```

```
from nltk.tokenize import word_tokenize

def tokenize_words(text):
    """Tokenizes a string into a list of words."""
    return word_tokenize(text)
```

```
from nltk.corpus import stopwords

def remove_stopwords(word_list):
    """Removes common stopwords from a list of words."""
    stop_words = set(stopwords.words('english'))
    return [word for word in word_list if word not in stop_words]
```

```
def add_sentence_tokens(sentences):
    """Adds start (<s>) and end (</s>) tokens to each sentence in a list of sen
    tokenized_sentences = []
    for sentence in sentences:
        tokenized_sentences.append(['<s>'] + word_tokenize(to_lowercase(remove_
    return tokenized_sentences
```

```
from nltk.tokenize import sent_tokenize

# Step 1: Split the cleaned corpus into sentences
sentences = sent_tokenize(cleaned_corpus)

# Step 2: Apply the add_sentence_tokens function
# This function internally applies to_lowercase, remove_punctuation_and_numbers
tokenized_sentences_with_tokens = add_sentence_tokens(sentences)

print("--- Processed Sentences with Start/End Tokens (first 3) ---")
for i, ts in enumerate(tokenized_sentences_with_tokens):
    if i < 3:
        print(f"{i+1}: {ts}")

print(f"\nTotal processed sentences: {len(tokenized_sentences_with_tokens)}")
```

```
--- Processed Sentences with Start/End Tokens (first 3) ---
1: ['<s>', 'this', 'is', 'the', 'first', 'sentence', 'of', 'our', 'sample', 'tex
2: ['<s>', 'it', 'discusses', 'natural', 'language', 'processing', '</s>']
3: ['<s>', 'ngrams', 'are', 'sequences', 'of', 'n', 'words', '</s>']

Total processed sentences: 7
```

```
def generate_ngrams(token_list, n):
    """Generates N-grams from a list of tokens."""
    ngrams = []
    for i in range(len(token_list) - n + 1):
        ngrams.append(tuple(token_list[i:i+n]))
    return ngrams

def calculate_conditional_probabilities(ngram_counts, preceding_counts):
    """Calculates conditional probabilities for N-grams."""
    probabilities = {}
    for ngram, count in ngram_counts.items():
        preceding_ngram = ngram[:-1]
        if preceding_ngram in preceding_counts and preceding_counts[preceding_ngr
            probabilities[ngram] = count / preceding_counts[preceding_ngram]
        else:
            probabilities[ngram] = 0 # Handle cases where preceding_ngram is not
    return probabilities
```

```
# Calculate the vocabulary size (V) based on all unique tokens in the corpus
# We'll use the unigram_counts for this, which includes start and end tokens.
vocabulary_size = len(unigram_counts)

def calculate_add_one_smoothed_probabilities(ngram_counts, preceding_counts, vo
```

```
    """Calculates conditional probabilities for N-grams using Add-one (Laplace)
    smoothed_probabilities = {}
    for ngram in ngram_counts:
        # Count(w1...wn)
        count_ngram = ngram_counts[ngram]
        # Count(w1...wn-1)
        preceding_ngram = ngram[:-1]
        count_preceding = preceding_counts.get(preceding_ngram, 0)

        # Smoothed probability = (Count(w1...wn) + 1) / (Count(w1...wn-1) + V)
        smoothed_probabilities[ngram] = (count_ngram + 1) / (count_preceding +

    # Also, we need to consider N-grams that might have a preceding part (w1...
    # but the full ngram (w1...wn) itself might not be in ngram_counts (count_n
    # These are effectively unseen combinations. Add-one smoothing handles this
    # to numerator and V to denominator if the ngram is not present, for any po
    # However, our current iteration only goes through existing ngram_counts. F
    # bigrams/trigrams, their individual calculation would still be 1 / (count_
    # For simplicity, we are calculating for observed ngrams and rely on 'get'
    # A more complete implementation might iterate over all possible (preceding

    return smoothed_probabilities

print(f"Calculated Vocabulary Size (V): {vocabulary_size}\n")

# --- Apply Add-one Smoothing to Bigram Model ---
smoothed_bigram_conditional_probabilities = calculate_add_one_smoothed_probabil
    bigram_counts,
    preceding_word_counts_for_calc,
    vocabulary_size
)

print("--- Smoothed Bigram Conditional Probabilities (Top 10) ---")
sorted_smoothed_bigram_probs = sorted(smoothed_bigram_conditional_probabilities
for bigram, prob in sorted_smoothed_bigram_probs[:10]:
    print(f"{bigram}: {prob:.4f}")

# --- Apply Add-one Smoothing to Trigram Model ---
smoothed_trigram_conditional_probabilities = calculate_add_one_smoothed_probabi
    trigram_counts,
    preceding_bigram_counts,
    vocabulary_size
)

print("\n--- Smoothed Trigram Conditional Probabilities (Top 10) ---")
sorted_smoothed_trigram_probs = sorted(smoothed_trigram_conditional_probabiliti
for trigram, prob in sorted_smoothed_trigram_probs[:10]:
    print(f"{trigram}: {prob:.4f}")
```

```
Calculated Vocabulary Size (V): 34

--- Smoothed Bigram Conditional Probabilities (Top 10) ---
('<s>', 'this'): 0.0732
('first', 'sentence'): 0.0571
('sentence', 'of'): 0.0571
('our', 'sample'): 0.0571
('it', 'discusses'): 0.0571
```

```
('discusses', 'natural'): 0.0571
('natural', 'language'): 0.0571
('language', 'processing'): 0.0571
('ngrams', 'are'): 0.0571
('are', 'sequences'): 0.0571

--- Smoothed Trigram Conditional Probabilities (Top 10) ---
('this', 'is', 'the'): 0.0571
('is', 'the', 'first'): 0.0571
('the', 'first', 'sentence'): 0.0571
('first', 'sentence', 'of'): 0.0571
('sentence', 'of', 'our'): 0.0571
('of', 'our', 'sample'): 0.0571
('our', 'sample', 'text'): 0.0571
('sample', 'text', '</s>'): 0.0571
('<s>', 'it', 'discusses'): 0.0571
('it', 'discusses', 'natural'): 0.0571
```

```
# Flatten the list of lists into a single list of tokens for Unigram and Bigram
all_tokens = [token for sentence_tokens in tokenized_sentences_with_tokens for

# --- Unigram Model ---
# Unigram counts (word counts)
unigram_counts = Counter(all_tokens)

# Unigram probabilities P(word) = count(word) / total_words
total_words = sum(unigram_counts.values())
# Exclude start/end tokens from total word count if desired for P(word), but in
# For simplicity here, we'll keep them in total_words for relative frequency fo
unigram_probabilities = {word: count / total_words for word, count in unigram_c

print("--- Unigram Counts (Top 10) ---")
for word, count in unigram_counts.most_common(10):
    print(f"{word}: {count}")

print("\n--- Unigram Probabilities (Top 10) ---")
# Sort by probability for display
sorted_unigram_probs = sorted(unigram_probabilities.items(), key=lambda item: i
for word, prob in sorted_unigram_probs[:10]:
    print(f"{word}: {prob:.4f}")
```

```
--- Unigram Counts (Top 10) ---
<s>: 7
</s>: 7
of: 4
is: 3
this: 2
the: 2
sample: 2
text: 2
processing: 2
first: 1

--- Unigram Probabilities (Top 10) ---
<s>: 0.1250
</s>: 0.1250
of: 0.0714
is: 0.0536
this: 0.0357
```

```
the: 0.0357
sample: 0.0357
text: 0.0357
processing: 0.0357
first: 0.0179
```

```
# --- Bigram Model ---
bigrams = []
for sentence_tokens in tokenized_sentences_with_tokens:
    bigrams.extend(generate_ngrams(sentence_tokens, 2))

bigram_counts = Counter(bigrams)

# Prepare preceding word counts for conditional probabilities (P(w2|w1))
# This means counting individual words (w1) when they appear as the start of a
# FIX: The keys of this counter need to be tuples (word,) to match ngram[:-1] f
preceding_word_counts_for_calc = Counter((token[0],) for token in bigrams)

# Conditional Probabilities P(word2 | word1)
bigram_conditional_probabilities = calculate_conditional_probabilities(
    bigram_counts,
    preceding_word_counts_for_calc # Use the corrected counts
)

print("--- Bigram Counts (Top 10) ---")
for bigram, count in bigram_counts.most_common(10):
    print(f"{bigram}: {count}")

print("\n--- Bigram Conditional Probabilities (Top 10) ---")
sorted_bigram_probs = sorted(bigram_conditional_probabilities.items(), key=lamb
for bigram, prob in sorted_bigram_probs[:10]:
    print(f"{bigram}: {prob:.4f}")
```

```
--- Bigram Counts (Top 10) ---
('<s>', 'this'): 2
('this', 'is'): 1
('is', 'the'): 1
('the', 'first'): 1
('first', 'sentence'): 1
('sentence', 'of'): 1
('of', 'our'): 1
('our', 'sample'): 1
('sample', 'text'): 1
('text', '</s>'): 1

--- Bigram Conditional Probabilities (Top 10) ---
('first', 'sentence'): 1.0000
('sentence', 'of'): 1.0000
('our', 'sample'): 1.0000
('it', 'discusses'): 1.0000
('discusses', 'natural'): 1.0000
('natural', 'language'): 1.0000
('language', 'processing'): 1.0000
('ngrams', 'are'): 1.0000
('are', 'sequences'): 1.0000
('sequences', 'of'): 1.0000
```

```
# --- Trigram Model ---
trigrams = []
for sentence_tokens in tokenized_sentences_with_tokens:
    trigrams.extend(generate_ngrams(sentence_tokens, 3))

trigram_counts = Counter(trigrams)

# Prepare preceding bigram counts for conditional probabilities (P(w3|w1, w2))
# This means counting bigrams (w1, w2) when they appear as the start of a trigr
preceding_bigram_counts = Counter(token[:-1] for token in trigrams)

# Conditional Probabilities P(word3 | word1, word2)
trigram_conditional_probabilities = calculate_conditional_probabilities(
    trigram_counts,
    preceding_bigram_counts
)

print("--- Trigram Counts (Top 10) ---")
for trigram, count in trigram_counts.most_common(10):
    print(f"{trigram}: {count}")

print("\n--- Trigram Conditional Probabilities (Top 10) ---")
sorted_trigram_probs = sorted(trigram_conditional_probabilities.items(), key=la
for trigram, prob in sorted_trigram_probs[:10]:
    print(f"{trigram}: {prob:.4f}")
```

```
--- Trigram Counts (Top 10) ---
('<s>', 'this', 'is'): 1
('this', 'is', 'the'): 1
('is', 'the', 'first'): 1
('the', 'first', 'sentence'): 1
('first', 'sentence', 'of'): 1
('sentence', 'of', 'our'): 1
('of', 'our', 'sample'): 1
('our', 'sample', 'text'): 1
('sample', 'text', '</s>'): 1
('<s>', 'it', 'discusses'): 1

--- Trigram Conditional Probabilities (Top 10) ---
('this', 'is', 'the'): 1.0000
('is', 'the', 'first'): 1.0000
('the', 'first', 'sentence'): 1.0000
('first', 'sentence', 'of'): 1.0000
('sentence', 'of', 'our'): 1.0000
('of', 'our', 'sample'): 1.0000
('our', 'sample', 'text'): 1.0000
('sample', 'text', '</s>'): 1.0000
('<s>', 'it', 'discusses'): 1.0000
('it', 'discusses', 'natural'): 1.0000
```

```
# Ensure total_words is defined for Unigram Smoothing
total_tokens = sum(unigram_counts.values())

def calculate_sentence_probability_unigram(sentence_tokens, unigram_counts, tot
    """Calculates the probability of a sentence using the Unigram model with Ad
    sentence_prob = 1.0
    for word in sentence_tokens:
        count_word = unigram_counts.get(word, 0)
```

```
        # Add-one smoothed unigram probability: (count(word) + 1) / (N + V)
        word_prob = (count_word + 1) / (total_tokens + vocabulary_size)
        sentence_prob *= word_prob
    return sentence_prob

def calculate_sentence_probability_bigram(sentence_tokens, bigram_counts, prece
    """Calculates the probability of a sentence using the Bigram model with Add
    sentence_prob = 1.0
    # Iterate through bigrams (w_prev, w_curr) in the sentence
    for i in range(len(sentence_tokens) - 1):
        w_prev = sentence_tokens[i]
        w_curr = sentence_tokens[i+1]

        bigram = (w_prev, w_curr)
        count_bigram = bigram_counts.get(bigram, 0)

        # Denominator for P(w_curr | w_prev): Count(w_prev) + V
        # The 'preceding_word_counts_for_calc' stores (w_prev,) as keys, so acc
        count_prev = preceding_word_counts_for_calc.get((w_prev,), 0)

        # Add-one smoothed bigram probability: (count(w_prev, w_curr) + 1) / (c
        bigram_prob = (count_bigram + 1) / (count_prev + vocabulary_size)
        sentence_prob *= bigram_prob
    return sentence_prob

def calculate_sentence_probability_trigram(sentence_tokens, trigram_counts, pre
    """Calculates the probability of a sentence using the Trigram model with Ad
    sentence_prob = 1.0
    # Iterate through trigrams (w_prev1, w_prev2, w_curr) in the sentence
    for i in range(len(sentence_tokens) - 2):
        w_prev1 = sentence_tokens[i]
        w_prev2 = sentence_tokens[i+1]
        w_curr = sentence_tokens[i+2]

        trigram = (w_prev1, w_prev2, w_curr)
        count_trigram = trigram_counts.get(trigram, 0)

        # Denominator for P(w_curr | w_prev1, w_prev2): Count(w_prev1, w_prev2)
        # The 'preceding_bigram_counts' stores (w_prev1, w_prev2) as keys
        count_prev_bigram = preceding_bigram_counts.get((w_prev1, w_prev2), 0)

        # Add-one smoothed trigram probability: (count(w_prev1, w_prev2, w_curr
        trigram_prob = (count_trigram + 1) / (count_prev_bigram + vocabulary_si
        sentence_prob *= trigram_prob
    return sentence_prob


print(f"--- Sentence Probabilities (Using Add-one Smoothed Models) ---")

# Choose at least 5 sentences for probability calculation
# We will use the already tokenized and preprocessed sentences from our corpus
sentences_to_test = tokenized_sentences_with_tokens[:5] # Using the first 5 pro

for idx, sentence in enumerate(sentences_to_test):
    print(f"\nSentence {idx+1}: {' '.join(sentence)}")
```

```
    # Unigram Probability
    prob_uni = calculate_sentence_probability_unigram(sentence, unigram_counts,
    print(f"  Unigram Probability: {prob_uni:.10e}") # Use scientific notation

    # Bigram Probability
    prob_bi = calculate_sentence_probability_bigram(sentence, bigram_counts, pr
    print(f"  Bigram Probability:  {prob_bi:.10e}")

    # Trigram Probability
    prob_tri = calculate_sentence_probability_trigram(sentence, trigram_counts,
    print(f"  Trigram Probability: {prob_tri:.10e}")
```

```
--- Sentence Probabilities (Using Add-one Smoothed Models) ---

Sentence 1: <s> this is the first sentence of our sample text </s>
  Unigram Probability: 2.6431229867e-16
  Bigram Probability:  3.7000526351e-13
  Trigram Probability: 6.3157234550e-12

Sentence 2: <s> it discusses natural language processing </s>
  Unigram Probability: 6.4227888577e-11
  Bigram Probability:  2.8894916188e-08
  Trigram Probability: 6.0926994705e-07

Sentence 3: <s> ngrams are sequences of n words </s>
  Unigram Probability: 2.3788106880e-12
  Bigram Probability:  1.5642360643e-09
  Trigram Probability: 3.4815425545e-08

Sentence 4: <s> another paragraph follows </s>
  Unigram Probability: 8.6707649579e-08
  Bigram Probability:  9.1018985992e-06
  Trigram Probability: 1.8658892128e-04

Sentence 5: <s> this one is about the importance of clean data </s>
  Unigram Probability: 1.1747213274e-16
  Bigram Probability:  3.9145046654e-13
  Trigram Probability: 6.3157234550e-12
```

```
def calculate_perplexity_unigram(sentence_tokens, unigram_counts, total_tokens,
    """Calculates the perplexity of a sentence using the Unigram model with Add
    # Perplexity is 1 / P(sentence)^(1/N), where N is the number of words in th
    # We need to handle log probabilities to avoid underflow with very small nu
    log_prob_sum = 0.0
    # The length N for perplexity is typically the number of actual words, excl
    # For this calculation, we'll exclude <s> and </s> from the N count.
    effective_sentence_length = len([token for token in sentence_tokens if toke

    if effective_sentence_length == 0:
        return float('inf') # Avoid division by zero

    for word in sentence_tokens:
        # The probability calculation is the same as in sentence probability
        count_word = unigram_counts.get(word, 0)
        word_prob = (count_word + 1) / (total_tokens + vocabulary_size)
        if word_prob > 0: # Avoid log(0)
            log_prob_sum += math.log(word_prob, 2) # Use base 2 for convention
        else:
```

```
            # If word_prob is 0 (should not happen with Add-one smoothing), it
            return float('inf')

    # Perplexity = 2 ^ (- (1/N) * log_prob_sum)
    perplexity = 2 ** (- (1 / effective_sentence_length) * log_prob_sum)
    return perplexity

def calculate_perplexity_bigram(sentence_tokens, bigram_counts, preceding_word_
    """Calculates the perplexity of a sentence using the Bigram model with Add-
    log_prob_sum = 0.0
    effective_sentence_length = len([token for token in sentence_tokens if toke

    if effective_sentence_length == 0:
        return float('inf')

    for i in range(len(sentence_tokens) - 1):
        w_prev = sentence_tokens[i]
        w_curr = sentence_tokens[i+1]

        bigram = (w_prev, w_curr)
        count_bigram = bigram_counts.get(bigram, 0)
        count_prev = preceding_word_counts_for_calc.get((w_prev,), 0)

        bigram_prob = (count_bigram + 1) / (count_prev + vocabulary_size)
        if bigram_prob > 0:
            log_prob_sum += math.log(bigram_prob, 2)
        else:
            return float('inf')

    # Note: For bigram perplexity, N is often considered the number of words *p
    # We'll use the same 'effective_sentence_length' as unigram for consistency
    perplexity = 2 ** (- (1 / effective_sentence_length) * log_prob_sum)
    return perplexity

def calculate_perplexity_trigram(sentence_tokens, trigram_counts, preceding_big
    """Calculates the perplexity of a sentence using the Trigram model with Add
    log_prob_sum = 0.0
    effective_sentence_length = len([token for token in sentence_tokens if toke

    if effective_sentence_length == 0:
        return float('inf')

    for i in range(len(sentence_tokens) - 2):
        w_prev1 = sentence_tokens[i]
        w_prev2 = sentence_tokens[i+1]
        w_curr = sentence_tokens[i+2]

        trigram = (w_prev1, w_prev2, w_curr)
        count_trigram = trigram_counts.get(trigram, 0)
        count_prev_bigram = preceding_bigram_counts.get((w_prev1, w_prev2), 0)

        trigram_prob = (count_trigram + 1) / (count_prev_bigram + vocabulary_si
        if trigram_prob > 0:
            log_prob_sum += math.log(trigram_prob, 2)
        else:
            return float('inf')
```

```
    # For trigram perplexity, N is often considered the number of words predict
    # We'll use the same 'effective_sentence_length' as unigram for consistency
    perplexity = 2 ** (- (1 / effective_sentence_length) * log_prob_sum)
    return perplexity


print(f"--- Sentence Perplexity (Using Add-one Smoothed Models) --- ")

# We will use the same sentences_to_test from the probability calculation step
# sentences_to_test = tokenized_sentences_with_tokens[:5]

for idx, sentence in enumerate(sentences_to_test):
    print(f"\nSentence {idx+1}: {' '.join(sentence)}")

    # Unigram Perplexity
    perp_uni = calculate_perplexity_unigram(sentence, unigram_counts, total_tok
    print(f"  Unigram Perplexity: {perp_uni:.4f}")

    # Bigram Perplexity
    perp_bi = calculate_perplexity_bigram(sentence, bigram_counts, preceding_wo
    print(f"  Bigram Perplexity:  {perp_bi:.4f}")

    # Trigram Perplexity
    perp_tri = calculate_perplexity_trigram(sentence, trigram_counts, preceding
    print(f"  Trigram Perplexity: {perp_tri:.4f}")
```

```
--- Sentence Perplexity (Using Add-one Smoothed Models) ---

Sentence 1: <s> this is the first sentence of our sample text </s>
  Unigram Perplexity: 53.8116
  Bigram Perplexity:  24.0608
  Trigram Perplexity: 17.5549

Sentence 2: <s> it discusses natural language processing </s>
  Unigram Perplexity: 109.2585
  Bigram Perplexity:  32.1985
  Trigram Perplexity: 17.5000

Sentence 3: <s> ngrams are sequences of n words </s>
  Unigram Perplexity: 86.5513
  Bigram Perplexity:  29.3506
  Trigram Perplexity: 17.5000

Sentence 4: <s> another paragraph follows </s>
  Unigram Perplexity: 225.9336
  Bigram Perplexity:  47.8949
  Trigram Perplexity: 17.5000

Sentence 5: <s> this one is about the importance of clean data </s>
  Unigram Perplexity: 58.8853
  Bigram Perplexity:  23.9107
  Trigram Perplexity: 17.5549
```