```
# Install nltk (only once)
!pip install nltk # Installs the Natural Language Toolkit library for NLP tasks
```

```
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) (1.5.3)
Requirement already satisfied: regex>=2021.8.3 in /usr/local/lib/python3.12/dist-packages (from nltk) (2025.11.3)
Requirement already satisfied: tqdm in /usr/local/lib/python3.12/dist-packages (from nltk) (4.67.1)
```

```
# Import required libraries
import nltk # Core NLTK library for natural language processing
import re # Regular expression operations for text cleaning
import math # Mathematical functions for calculations like log and exp (used in perplexity)
import random # For shuffling data (e.g., sentences for train/test split)
import numpy as np # Numerical operations, often used with arrays (though not directly used in these snippets)
import pandas as pd # Data manipulation and analysis (though not directly used in these snippets)
from collections import Counter, defaultdict # Specialized container datatypes, Counter for frequency counting of n-grams
from nltk.tokenize import sent_tokenize, word_tokenize # Functions for splitting text into sentences and words
from nltk.corpus import stopwords # Access to a list of common stopwords for filtering
```

```
# Download required NLTK resources
nltk.download('punkt') # Downloads the 'punkt' tokenizer models, essential for sentence tokenization
nltk.download('stopwords') # Downloads the list of common stopwords for various languages (e.g., English)
```

```
[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!
True
```

```
# Sample text corpus (repeat to reach 1500+ words)
text = """
Natural Language Processing is a field of artificial intelligence.
It helps computers understand human language.
Language models predict the probability of words.
Unigram, Bigram and Trigram models are basic models.
They are widely used in NLP applications.
""" * 300   # Repeats the base text 300 times to create a large corpus (exceeding 1500 words for better training data)
```

```
# Display sample text
```

```
print(text[:500]) # Prints the first 500 characters of the generated text corpus to show its content
```

```
Natural Language Processing is a field of artificial intelligence.
It helps computers understand human language.
Language models predict the probability of words.
Unigram, Bigram and Trigram models are basic models.
They are widely used in NLP applications.

Natural Language Processing is a field of artificial intelligence.
It helps computers understand human language.
Language models predict the probability of words.
Unigram, Bigram and Trigram models are basic models.
They are widely used in
```

```
def preprocess_text(text):
    """
    Preprocesses the input text by lowercasing, removing punctuation and numbers,
    tokenizing into sentences and words, and removing stopwords. Each sentence
    is then padded with start ('<s>') and end ('</s>') tokens.

    Args:
        text (str): The raw input text corpus.

    Returns:
        list: A list of lists, where each inner list represents a processed sentence
              with start and end tokens, and no stopwords or punctuation.
    """
    text = text.lower()                      # Convert the entire input text to lowercase for case-insensitivity
    text = re.sub(r'[^a-z\s]', '', text)     # Remove all characters that are not lowercase English letters or spaces
    sentences = sent_tokenize(text)          # Tokenize the cleaned text into a list of individual sentences

    stop_words = set(stopwords.words('english')) # Load English stopwords into a set for efficient checking
    processed_sentences = []                 # Initialize an empty list to store the final processed sentences

    for sent in sentences:                   # Iterate through each tokenized sentence
        words = word_tokenize(sent)          # Tokenize the current sentence string into a list of words
        words = [w for w in words if w not in stop_words] # Filter out stopwords from the list of words
        processed_sentences.append(['<s>'] + words + ['</s>']) # Prepend '<s>' and append '</s>' tokens to the word list, the

    return processed_sentences               # Return the list of all processed sentences
```

```
nltk.download('punkt_tab') # Download the punkt_tab tokenizer model for NLTK, which might be needed internally by sent_tokeni
sentences = preprocess_text(text) # Preprocess the main text corpus using the defined function into a list of tokenized sente
print(sentences[:2]) # Display the first two preprocessed sentences to verify the output format and content
```

```
[['<s>', 'natural', 'language', 'processing', 'field', 'artificial', 'intelligence', 'helps', 'computers', 'understand', 'huma
[nltk_data] Downloading package punkt_tab to /root/nltk_data...
[nltk_data]   Package punkt_tab is already up-to-date!
```

```
random.shuffle(sentences) # Randomly shuffles the order of the sentences to ensure a diverse distribution for training and te

split = int(0.8 * len(sentences)) # Determine the split point for an 80/20 train/test split (80% for training)
train_sentences = sentences[:split] # Assign the first 80% of the shuffled sentences to the training set
test_sentences = sentences[split:] # Assign the remaining 20% of the shuffled sentences to the test set
```

```
def build_ngram(sentences, n):
    """
    Builds n-grams from a list of sentences.

    Args:
        sentences (list): A list of tokenized sentences.
        n (int): The size of the n-gram (e.g., 1 for unigram, 2 for bigram).

    Returns:
        collections.Counter: A Counter object containing the frequency of each n-gram.
    """
    ngrams = []
    for sent in sentences:
        # Iterate through each sentence to extract n-grams
        # zip(*[sent[i:] for i in range(n)]) creates tuples of n consecutive words
        ngrams += list(zip(*[sent[i:] for i in range(n)]))
    return Counter(ngrams) # Return the frequency count of each n-gram
```

```
unigram_counts = build_ngram(train_sentences, 1) # Build a Counter of all unigrams (single words) from the training sentences
bigram_counts = build_ngram(train_sentences, 2) # Build a Counter of all bigrams (two-word sequences) from the training sente
trigram_counts = build_ngram(train_sentences, 3) # Build a Counter of all trigrams (three-word sequences) from the training s

vocab_size = len(unigram_counts) # Calculate the size of the vocabulary (number of unique unigrams) from the training data
```

```
def sentence_probability(sentence, ngram_counts, lower_counts, n):
    """
```

```
    Calculates the probability of a given sentence using an n-gram model with Laplace smoothing.

    Args:
        sentence (list): A list of words representing the sentence (without start/end tokens).
        ngram_counts (collections.Counter): Counts of n-grams from the training data.
        lower_counts (collections.Counter): Counts of (n-1)-grams (context) from the training data.
                                            For unigrams, this can be an empty Counter.
        n (int): The order of the n-gram model (e.g., 1 for unigram, 2 for bigram, 3 for trigram).

    Returns:
        float: The probability of the sentence.
    """
    prob = 1 # Initialize the total probability of the sentence to 1 (multiplicative identity)
    sentence = ['<s>'] + sentence + ['</s>'] # Pad the input sentence with start and end tokens for consistent n-gram extract

    for i in range(len(sentence) - n + 1): # Iterate through the padded sentence to extract all possible n-grams
        ngram = tuple(sentence[i:i+n]) # Extract the current n-gram as a tuple
        lower_ngram = tuple(sentence[i:i+n-1]) # Extract the (n-1)-gram as the context for the current n-gram

        # Apply Laplace smoothing (add-1 smoothing) to handle unseen n-grams and avoid zero probabilities
        numerator = ngram_counts.get(ngram, 0) + 1 # Get count of current n-gram (0 if not found) and add 1 for smoothing
        denominator = lower_counts.get(lower_ngram, 0) + vocab_size # Get count of context n-gram (0 if not found) and add vo

        prob *= numerator / denominator # Multiply the total probability by the conditional probability of the current n-gram
    return prob # Return the final calculated probability of the entire sentence
```

```
# ---------- MINIMAL ONE CELL N-GRAM PROBABILITY CODE ----------

from collections import Counter # Import Counter for efficient frequency counting of items
import math # Import math for mathematical operations like log and exp

# Small sample training data (already tokenized for demonstration purposes)
train_sentences = [
    ['<s>', 'natural', 'language', 'processing', '</s>'],
    ['<s>', 'language', 'models', 'are', 'useful', '</s>'],
    ['<s>', 'ngram', 'models', 'predict', 'words', '</s>'],
    ['<s>', 'probability', 'is', 'important', '</s>'],
    ['<s>', 'language', 'processing', 'uses', 'models', '</s>']
]

# Small sample test data (already tokenized for demonstration)
test_sentences = [
```

```
    ['<s>', 'language', 'models', 'predict', 'words', '</s>'],
    ['<s>', 'natural', 'language', 'models', '</s>']
]

# Build n-grams function (redefined for this minimal example to be self-contained)
def build_ngram(sentences, n):
    ngrams = [] # Initialize an empty list to collect all n-grams
    for sent in sentences: # Iterate through each sentence in the provided list
        for i in range(len(sent) - n + 1): # Loop to extract all possible n-grams from the current sentence
            ngrams.append(tuple(sent[i:i+n])) # Append the extracted n-gram as a tuple to the list
    return Counter(ngrams) # Return a Counter object containing the frequency of each unique n-gram

unigram_counts = build_ngram(train_sentences, 1) # Build a Counter of all unigrams from the sample training sentences
bigram_counts = build_ngram(train_sentences, 2) # Build a Counter of all bigrams from the sample training sentences
trigram_counts = build_ngram(train_sentences, 3) # Build a Counter of all trigrams from the sample training sentences

vocab_size = len(unigram_counts) # Calculate the vocabulary size based on the number of unique unigrams in the training data

# Sentence probability with Laplace smoothing function (redefined for this minimal example)
def sentence_probability(sentence, ngram_counts, lower_counts, n):
    prob = 1 # Initialize the total probability of the sentence to 1
    for i in range(len(sentence) - n + 1): # Iterate through the sentence to extract all relevant n-grams
        ngram = tuple(sentence[i:i+n]) # Extract the current n-gram
        lower_ngram = tuple(sentence[i:i+n-1]) # Extract the (n-1)-gram, which serves as the context
        # Apply Laplace (add-1) smoothing: (count(ngram) + 1) / (count(context) + vocab_size)
        prob *= (ngram_counts.get(ngram, 0) + 1) / (lower_counts.get(lower_ngram, 0) + vocab_size)
    return prob # Return the calculated probability of the sentence

# Test sentences from the sample test data
test_sentences_sample = test_sentences[:5] # Select the first 5 sentences from the small test set for demonstration

for sent in test_sentences_sample: # Iterate through each selected test sentence
    words = sent[1:-1] # Extract the actual words from the sentence, excluding the '<s>' and '</s>' tokens for display
    print("Sentence:", " ".join(words)) # Print the current test sentence
    # Calculate and print probabilities for Unigram, Bigram, and Trigram models using the sentence_probability function
    print("Unigram Prob:", sentence_probability(sent, unigram_counts, Counter(), 1)) # Probability using unigram model
    print("Bigram Prob :", sentence_probability(sent, bigram_counts, unigram_counts, 2)) # Probability using bigram model
    print("Trigram Prob:", sentence_probability(sent, trigram_counts, bigram_counts, 3)) # Probability using trigram model
    print("-" * 40) # Print a separator line for better readability between sentences
```

```
Sentence: language models predict words
Unigram Prob: 0.00020227160493827164
Bigram Prob : 2.8935185185185183e-05
Trigram Prob: 0.00011488970588235294
```

```
----------------------------------------
Sentence: natural language models
Unigram Prob: 0.0015170370370370372
Bigram Prob : 0.00015432098765432098
Trigram Prob: 0.00048828125
----------------------------------------
```

```
def perplexity(sentence, ngram_counts, lower_counts, n):
    """
    Calculates the perplexity of a given sentence using an n-gram model with Laplace smoothing.
    Perplexity is a measure of how well a probability distribution predicts a sample. A lower perplexity
    indicates a better language model.

    Args:
        sentence (list): A list of words representing the sentence (without start/end tokens).
        ngram_counts (collections.Counter): Counts of n-grams from the training data.
        lower_counts (collections.Counter): Counts of (n-1)-grams (context) from the training data.
                                            For unigrams, this can be an empty Counter.
        n (int): The order of the n-gram model.

    Returns:
        float: The perplexity score of the sentence.
    """
    sentence = ['<s>'] + sentence + ['</s>'] # Pad the input sentence with start and end tokens for consistent n-gram extract
    N = len(sentence) # Total number of tokens in the padded sentence, used for normalization in perplexity formula
    log_prob = 0 # Initialize the cumulative sum of natural logarithms of probabilities

    for i in range(len(sentence) - n + 1): # Iterate through the padded sentence to extract all possible n-grams
        ngram = tuple(sentence[i:i+n]) # Extract the current n-gram as a tuple
        lower_ngram = tuple(sentence[i:i+n-1]) # Extract the (n-1)-gram (context) for the current n-gram

        # Calculate probability with Laplace smoothing to avoid zero probabilities
        # Note: vocab_size should be correctly calculated from the unigram counts of the training data
        prob = (ngram_counts.get(ngram, 0) + 1) / (lower_counts.get(lower_ngram, 0) + vocab_size) # Conditional probability w
        log_prob += math.log(prob) # Add the natural logarithm of the calculated probability to the sum

    # Calculate perplexity using the formula: exp(- (sum of log probabilities) / N)
    return math.exp(-log_prob / N) # Return the calculated perplexity score
```

```
for sent in test_sentences_sample: # Loop through each sample test sentence
    words = sent[1:-1] # Extract words excluding start/end tokens for printing purposes
    print("Sentence:", " ".join(words)) # Print the current test sentence
```

```
    # Calculate and print perplexity for Unigram, Bigram, and Trigram models using the perplexity function
    print("Unigram PP:", perplexity(words, unigram_counts, Counter(), 1)) # Perplexity using unigram model
    print("Bigram PP :", perplexity(words, bigram_counts, unigram_counts, 2)) # Perplexity using bigram model
    print("Trigram PP:", perplexity(words, trigram_counts, bigram_counts, 3)) # Perplexity using trigram model
    print() # Print an empty line for better readability between the output of different sentences
```

```
Sentence: language models predict words
Unigram PP: 4.127409061118283
Bigram PP : 5.707277056455109
Trigram PP: 4.535444049403089

Sentence: natural language models
Unigram PP: 3.662695064479402
Bigram PP : 5.785155024015764
Trigram PP: 4.594793419988139
```