N-Gram Language Model Implementation and Evaluation - Perplexity

Libraries for N-Gram Language Model

`nltk` (Natural Language Toolkit):

- **Purpose:** `nltk` is a powerful library for working with human language data. It provides easy-to-use interfaces to over 50 corpora and lexical resources, along with a suite of text processing libraries for classification, tokenization, stemming, tagging, parsing, and semantic reasoning.
- **Usage in this task:** Primarily used for text preprocessing tasks like tokenization (breaking text into words or sentences) which is crucial for building N-grams.

`numpy`:

- **Purpose:** `numpy` is the fundamental package for numerical computation in Python. It provides support for large, multi-dimensional arrays and matrices, along with a collection of high-level mathematical functions to operate on these arrays.
- **Usage in this task:** Useful for various numerical operations, especially when dealing with probabilities, calculations related to N-gram frequencies, and potentially for more complex statistical analyses.

`pandas`:

- **Purpose:** `pandas` is a fast, powerful, flexible, and easy-to-use open-source data analysis and manipulation tool, built on top of the Python programming language.
- **Usage in this task:** While not strictly necessary for basic N-gram counting, `pandas` can be incredibly useful for managing and analyzing the N-gram frequencies, probabilities, and for organizing any tabular data derived from the text analysis.

`matplotlib.pyplot`:

- **Purpose:** `matplotlib.pyplot` is a plotting library that provides a MATLAB-like way of plotting. It is often used with `numpy` to create a variety of static, animated, and interactive visualizations in Python.
- **Usage in this task:** If we decide to visualize N-gram distributions, frequency plots, or perplexity results, `matplotlib.pyplot` will be the go-to tool.

```
nltk.download('punkt')
nltk.download('punkt_tab')

# Tokenize the text into words
# Explanation for nltk.word_tokenize:
# nltk.word_tokenize is a function from the Natural Language Toolkit
(NLTK) library
# that splits a string into a list of words and punctuation marks.
# It's a robust tokenizer that handles various cases like contractions
and punctuation attached to words.
```

```
words = nltk.word_tokenize(corpus_text)

# Convert words to lowercase and remove non-alphabetic tokens
# Explanation for lowercasing and filtering:
# Converting words to lowercase ensures that words like 'The' and
'the' are treated as the same, reducing vocabulary size
# and improving model generalization. Filtering for isalpha() removes
punctuation, numbers, and other non-alphabetic tokens,
# which are typically not desired in basic N-gram language models as
they don't contribute to word sequence probabilities.
processed_words = [word.lower() for word in words if word.isalpha()]

print(f"Original token count: {len(words)}")
print(f"Processed token count (lowercase, alpha-only):
{len(processed_words)}")
print("\nFirst 20 processed words:")
print(processed_words[:20])

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

Original token count: 1175
Processed token count (lowercase, alpha-only): 842

First 20 processed words:
['it', 'was', 'the', 'first', 'day', 'of', 'the', 'semester', 'at',
'sunrise', 'institute', 'of', 'technology', 'the', 'campus', 'looked',
'fresh', 'after', 'the', 'rain']

import sys
!{sys.executable} -m pip install python-docx

Collecting python-docx
  Downloading python_docx-1.2.0-py3-none-any.whl.metadata (2.0 kB)
Requirement already satisfied: lxml>=3.1.0 in
/usr/local/lib/python3.12/dist-packages (from python-docx) (6.0.2)
Requirement already satisfied: typing_extensions>=4.9.0 in
/usr/local/lib/python3.12/dist-packages (from python-docx) (4.15.0)
Downloading python_docx-1.2.0-py3-none-any.whl (252 kB)
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0.0/253.0 kB ? eta -:--:--
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 253.0/253.0 kB 8.4 MB/s eta
0:00:00

import docx

def read_docx(file_path):
    doc = docx.Document(file_path)
    full_text = []
    for para in doc.paragraphs:
        full_text.append(para.text)
    return '\n'.join(full_text)

docx_file_path = '/content/N-Gram Language Model Implementation and
Evaluation -DATASET.docx'
corpus_text = read_docx(docx_file_path)

print(f"Successfully extracted {len(corpus_text)} characters from the
document.\n")
print("First 500 characters of the extracted text:")
print("-" * 50)
print(corpus_text[:500])
print("-" * 50)
```

```
Successfully extracted 5449 characters from the document.

First 500 characters of the extracted text:
--------------------------------------------------
It was the first day of the semester at Sunrise Institute of
Technology. The campus looked fresh after the rain. The trees were
green. The sky was clear. The students were excited.
Arjun walked into the classroom slowly. He looked around and smiled.
He loved the smell of new books and clean boards. He loved the sound
of students talking about projects.
“Hey Arjun!” said Meera.
“Hey Meera! Are you ready for machine learning?” Arjun asked.
“I am ready, but I am also nervous,” Meera replied.
The pr
--------------------------------------------------
```

```
# Import collections.Counter for efficient counting of N-grams
from collections import Counter

# Explanation for Counter:
# Counter is a dict subclass for counting hashable objects. It's an
unordered collection
# where elements are stored as dictionary keys and their counts are
stored as dictionary values.
# This makes it ideal for easily counting the occurrences of words, N-
grams, or any sequence of items
# within a text dataset, which is a fundamental step in building an N-
gram language model.
```

This dataset is a fictional story set in a university campus focusing on students learning programming and machine learning. It combines narrative storytelling with natural conversational dialogues between characters. The text contains repeated linguistic patterns and varied vocabulary to support N-Gram probability learning. It is suitable for training unigram, bigram, and trigram language models. The dataset can be used to compute perplexity for evaluation. It also supports experimentation with smoothing techniques like Laplace smoothing.

Text Preprocessing Functions

To prepare our text data for N-gram modeling, we'll create several functions that perform the following steps:

1. **Tokenization:** Breaking down the text into individual words or sentences.
2. **Lowercasing:** Converting all words to lowercase to treat words like 'The' and 'the' as identical.
3. **Punctuation and Number Removal:** Filtering out non-alphabetic characters that don't contribute to the linguistic patterns we want to model.
4. **Optional Stopword Removal:** Eliminating common words (e.g., 'the', 'is', 'a') that often carry less semantic meaning and can be removed for certain N-gram analyses.
5. **Adding Start/End Tokens:** Inserting special markers (e.g., `<s>` for start, `</s>` for end) at the beginning and end of each sentence. This is crucial for N-gram models to learn probabilities at sentence boundaries and predict the start or end of a sequence.

```python
import nltk
from nltk.corpus import stopwords

nltk.download('stopwords')

# Explanation for NLTK Stopwords:
# Stopwords are common words (like 'the', 'is', 'in', 'a') that appear 
very frequently
# in a language but often do not carry significant meaning for certain 
NLP tasks.
# Removing them can reduce the vocabulary size and focus the N-gram 
model on more
# meaningful word sequences, though it's an optional step depending on 
the task.
stop_words = set(stopwords.words('english'))

def preprocess_sentence(sentence, remove_stopwords=False):
    """
    Preprocesses a single sentence: tokenizes, lowercases, removes 
non-alphabetic,
    optionally removes stopwords, and adds start/end tokens.
    """
    # Tokenize words
    # Explanation: Breaks the sentence into individual words and 
punctuation.
    tokens = nltk.word_tokenize(sentence)

    # Convert to lowercase and remove non-alphabetic
    # Explanation: Converts all tokens to lowercase to ensure 
consistency (e.g., 'Word' and 'word' are treated as same).
    # It then filters out any tokens that are not purely alphabetic
```

```
(e.g., numbers, punctuation).
    processed_tokens = [word.lower() for word in tokens if
word.isalpha()]

    # Optionally remove stopwords
    # Explanation: If 'remove_stopwords' is True, this step filters
out common words
    # from the NLTK English stopwords list, reducing noise for some
language modeling tasks.
    if remove_stopwords:
        processed_tokens = [word for word in processed_tokens if word
not in stop_words]

    # Add start and end tokens
    # Explanation: Adds special markers '<s>' at the beginning and
'</s>' at the end
    # of each sentence. This helps the N-gram model learn sentence
boundaries and
    # improve prediction of the first and last words of a sentence.
    return ['<s>'] + processed_tokens + ['</s>']

def preprocess_corpus(corpus_text, remove_stopwords=False):
    """
    Preprocesses the entire corpus: splits into sentences and applies
preprocess_sentence to each.
    """
    # Tokenize the corpus into sentences
    # Explanation: Splits the larger text into individual sentences.
This is important
    # for correctly applying start and end tokens to each distinct
sentence.
    sentences = nltk.sent_tokenize(corpus_text)

    all_processed_words = []
    for sent in sentences:
        all_processed_words.extend(preprocess_sentence(sent,
remove_stopwords))

    return all_processed_words

print("Preprocessing functions defined.")
```

```
Preprocessing functions defined.
```

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

```
# Example usage of the preprocessing functions
print("\n--- Preprocessing without Stopword Removal ---")
processed_corpus_no_stopwords = preprocess_corpus(corpus_text,
remove_stopwords=False)
```

```python
print(f"Total processed tokens (no stopwords):
{len(processed_corpus_no_stopwords)}")
print("First 30 tokens:")
print(processed_corpus_no_stopwords[:30])

print("\n--- Preprocessing with Stopword Removal ---")
processed_corpus_with_stopwords = preprocess_corpus(corpus_text,
remove_stopwords=True)
print(f"Total processed tokens (with stopwords):
{len(processed_corpus_with_stopwords)}")
print("First 30 tokens:")
print(processed_corpus_with_stopwords[:30])
```

```
--- Preprocessing without Stopword Removal ---
Total processed tokens (no stopwords): 1098
First 30 tokens:
['<s>', 'it', 'was', 'the', 'first', 'day', 'of', 'the', 'semester',
'at', 'sunrise', 'institute', 'of', 'technology', '</s>', '<s>',
'the', 'campus', 'looked', 'fresh', 'after', 'the', 'rain', '</s>',
'<s>', 'the', 'trees', 'were', 'green', '</s>']

--- Preprocessing with Stopword Removal ---
Total processed tokens (with stopwords): 812
First 30 tokens:
['<s>', 'first', 'day', 'semester', 'sunrise', 'institute',
'technology', '</s>', '<s>', 'campus', 'looked', 'fresh', 'rain',
'</s>', '<s>', 'trees', 'green', '</s>', '<s>', 'sky', 'clear',
'</s>', '<s>', 'students', 'excited', '</s>', '<s>', 'arjun',
'walked', 'classroom']
```

now we pre processed the text document

lets build N- GRAM MODEL

```python
import nltk
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

Task

Build Unigram, Bigram, and Trigram language models from the `processed_corpus_no_stopwords` list. For each model, calculate the counts and probabilities of the N-grams. Finally, present these models as pandas DataFrames, showing the N-grams, their counts, and their calculated probabilities.

Build Unigram Model and Counts

Subtask:

Construct the Unigram model by counting the frequency of each individual word in the processed corpus. Create a table showing each word and its count.

Reasoning: To construct the unigram model, I will use the `Counter` object to get the frequency of each word in the `processed_corpus_no_stopwords` list, as instructed. Then, I will convert these counts into a pandas DataFrame.

```
from collections import Counter

# Count the occurrences of each word in the processed corpus without
stopwords
unigram_counts = Counter(processed_corpus_no_stopwords)

# Convert the unigram counts into a pandas DataFrame
unigram_df = pd.DataFrame(unigram_counts.items(), columns=['Word',
'Count'])

# Display the first few rows of the DataFrame
print("Unigram Counts (Top 10):")
print(unigram_df.head(10))

# Display basic statistics about the unigram counts
print(f"\nTotal unique words (vocabulary size): {len(unigram_df)}")
print(f"Total word occurrences: {unigram_df['Count'].sum()}")
```

```
Unigram Counts (Top 10):
       Word  Count
0       <s>    128
1        it      7
2       was      4
3       the     96
4     first      2
5       day      3
6        of     10
7  semester      3
8        at      6
9   sunrise      1

Total unique words (vocabulary size): 304
Total word occurrences: 1098
```

Reasoning: Now that the unigram counts have been generated, the next step is to calculate the probability of each unigram. This is done by dividing each word's count by the total number of words in the corpus.

```
total_words = unigram_df['Count'].sum()
unigram_df['Probability'] = unigram_df['Count'] / total_words

# Sort by probability in descending order for better inspection
unigram_df = unigram_df.sort_values(by='Probability',
ascending=False).reset_index(drop=True)

print("Unigram Model with Probabilities (Top 10):")
print(unigram_df.head(10))
print(f"\nSum of unigram probabilities:
{unigram_df['Probability'].sum():.4f}")
```

```
Unigram Model with Probabilities (Top 10):
    Word  Count  Probability
0   </s>    128     0.116576
1    <s>    128     0.116576
2    the     96     0.087432
3  arjun     29     0.026412
4   they     23     0.020947
5   said     22     0.020036
6  meera     21     0.019126
7  model     20     0.018215
8      a     14     0.012750
9     is     12     0.010929

Sum of unigram probabilities: 1.0000
```

Build Bigram Model and Counts

Subtask:

Construct the Bigram model by identifying and counting the frequency of each two-word sequence (bigram) in the processed corpus. Create a table showing each bigram and its count.

Reasoning: To construct the bigram model, I will generate all possible two-word sequences (bigrams) from the `processed_corpus_no_stopwords` list. Then, I will use the `Counter` object to count the frequency of each bigram and store these counts in a pandas DataFrame.

```
from collections import Counter

# Generate bigrams from the processed corpus
# A bigram is a sequence of two consecutive words. We iterate through
the list
# and create tuples of (word_i, word_i+1).
bigrams = [(processed_corpus_no_stopwords[i],
processed_corpus_no_stopwords[i+1])
           for i in range(len(processed_corpus_no_stopwords) - 1)]

# Count the occurrences of each bigram
```

```
bigram_counts = Counter(bigrams)

# Convert the bigram counts into a pandas DataFrame
bigram_df = pd.DataFrame(bigram_counts.items(), columns=['Bigram',
'Count'])

# Display the first few rows of the DataFrame
print("Bigram Counts (Top 10):")
print(bigram_df.head(10))

# Display basic statistics about the bigram counts
print(f"\nTotal unique bigrams: {len(bigram_df)}")
print(f"Total bigram occurrences: {bigram_df['Count'].sum()}")

Bigram Counts (Top 10):
             Bigram  Count
0         (<s>, it)      5
1         (it, was)      1
2        (was, the)      1
3      (the, first)      1
4      (first, day)      1
5         (day, of)      1
6         (of, the)      2
7   (the, semester)      2
8    (semester, at)      1
9     (at, sunrise)      1

Total unique bigrams: 673
Total bigram occurrences: 1097
```

Reasoning: Now that the bigram counts have been generated, the next step is to calculate the probability of each bigram. This will involve dividing the bigram count by the count of the first word in the bigram (the unigram count of the context word).

```
merged_df = pd.merge(bigram_df, unigram_df,
left_on=bigram_df['Bigram'].apply(lambda x: x[0]), right_on='Word',
how='left')
merged_df['Probability'] = merged_df['Count_x'] / merged_df['Count_y']

bigram_model_df = merged_df[['Bigram', 'Count_x', 'Probability']]
bigram_model_df.columns = ['Bigram', 'Count', 'Probability']

# Sort by probability in descending order for better inspection
bigram_model_df = bigram_model_df.sort_values(by='Probability',
ascending=False).reset_index(drop=True)

print("Bigram Model with Probabilities (Top 10):")
print(bigram_model_df.head(10))
```

```
Bigram Model with Probabilities (Top 10):
                     Bigram  Count  Probability
0           (warmly, </s>)      1          1.0
1         (glowed, warmly)      1          1.0
2     (sunrise, institute)      1          1.0
3          (institute, of)      1          1.0
4              (story, of)      3          1.0
5          (suddenly, the)      1          1.0
6   (corrected, mistakes)      1          1.0
7            (tested, the)      2          1.0
8            (typed, code)      1          1.0
9             (helps, the)      2          1.0
```

Build Trigram Model and Counts

Subtask:

Construct the Trigram model by identifying and counting the frequency of each three-word sequence (trigram) in the processed corpus. Create a table showing each trigram and its count.

Reasoning: To construct the trigram model, I will generate all possible three-word sequences (trigrams) from the `processed_corpus_no_stopwords` list. Then, I will use the `Counter` object to count the frequency of each trigram and store these counts in a pandas DataFrame.

```python
from collections import Counter

# Generate trigrams from the processed corpus
# A trigram is a sequence of three consecutive words.
trigrams = [(processed_corpus_no_stopwords[i],
processed_corpus_no_stopwords[i+1],
processed_corpus_no_stopwords[i+2])
            for i in range(len(processed_corpus_no_stopwords) - 2)]

# Count the occurrences of each trigram
trigram_counts = Counter(trigrams)

# Convert the trigram counts into a pandas DataFrame
trigram_df = pd.DataFrame(trigram_counts.items(), columns=['Trigram',
'Count'])

# Display the first few rows of the DataFrame
print("Trigram Counts (Top 10):")
print(trigram_df.head(10))

# Display basic statistics about the trigram counts
print(f"\nTotal unique trigrams: {len(trigram_df)}")
print(f"Total trigram occurrences: {trigram_df['Count'].sum()}")
```

```
Trigram Counts (Top 10):
                       Trigram  Count
0             (<s>, it, was)      1
1             (it, was, the)      1
2          (was, the, first)      1
3          (the, first, day)      1
4           (first, day, of)      1
5             (day, of, the)      1
6        (of, the, semester)      1
7        (the, semester, at)      1
8    (semester, at, sunrise)      1
9  (at, sunrise, institute)      1

Total unique trigrams: 864
Total trigram occurrences: 1096
```

Reasoning: Now that the trigram counts have been generated, the next step is to calculate the probability of each trigram. This will involve dividing the trigram count by the count of the first two words in the trigram (the bigram count of the context words).

```
merged_trigram_df = pd.merge(trigram_df, bigram_df,
left_on=trigram_df['Trigram'].apply(lambda x: (x[0], x[1])),
right_on='Bigram', how='left')
merged_trigram_df['Probability'] = merged_trigram_df['Count_x'] /
merged_trigram_df['Count_y']

trigram_model_df = merged_trigram_df[['Trigram', 'Count_x',
'Probability']]
trigram_model_df.columns = ['Trigram', 'Count', 'Probability']

# Sort by probability in descending order for better inspection
trigram_model_df = trigram_model_df.sort_values(by='Probability',
ascending=False).reset_index(drop=True)

print("Trigram Model with Probabilities (Top 10):")
print(trigram_model_df.head(10))

Trigram Model with Probabilities (Top 10):
                            Trigram  Count  Probability
0  (friendship, continued, </s>)      1          1.0
1    (of, friendship, continued)      1          1.0
2      (coding, continued, </s>)      1          1.0
3        (of, coding, continued)      1          1.0
4    (learning, continued, </s>)      1          1.0
5      (of, learning, continued)      1          1.0
6                (the, story, of)      3          1.0
7             (warmly, </s>, <s>)      1          1.0
8          (glowed, warmly, </s>)      1          1.0
9         (lights, glowed, warmly)      1          1.0
```

Display N-gram Tables

Subtask:

Present the generated tables for Unigrams, Bigrams, and Trigrams, clearly showing their counts and conditional probabilities. Use pandas DataFrames for structured display.

Reasoning: The subtask requires displaying the previously generated DataFrames for unigrams, bigrams, and trigrams. I will use `print()` statements to show each DataFrame along with a descriptive label as instructed.

```
print("Unigram Model Table:")
print(unigram_df)

print("\nBigram Model Table:")
print(bigram_model_df)

print("\nTrigram Model Table:")
print(trigram_model_df)
```

```
Unigram Model Table:
         Word  Count  Probability
0        </s>    128     0.116576
1         <s>    128     0.116576
2         the     96     0.087432
3       arjun     29     0.026412
4        they     23     0.020947
..        ...    ...          ...
299  powerful      1     0.000911
300   predict      1     0.000911
301   improve      1     0.000911
302      data      1     0.000911
303      wind      1     0.000911

[304 rows x 3 columns]

Bigram Model Table:
                   Bigram  Count  Probability
0          (warmly, </s>)      1     1.000000
1        (glowed, warmly)      1     1.000000
2    (sunrise, institute)      1     1.000000
3         (institute, of)      1     1.000000
4             (story, of)      3     1.000000
..                    ...    ...          ...
668         (<s>, weeks)      1     0.007812
669           (<s>, let)      1     0.007812
670         (<s>, first)      1     0.007812
671         (<s>, today)      1     0.007812
672      (<s>, language)      1     0.007812
```

```
[673 rows x 3 columns]

Trigram Model Table:
                              Trigram  Count  Probability
0    (friendship, continued, </s>)      1     1.000000
1      (of, friendship, continued)      1     1.000000
2        (coding, continued, </s>)      1     1.000000
3          (of, coding, continued)      1     1.000000
4      (learning, continued, </s>)      1     1.000000
..                             ...    ...          ...
859                (</s>, <s>, why)      1     0.007874
860          (</s>, <s>, variation)      1     0.007874
861           (</s>, <s>, language)      1     0.007874
862              (</s>, <s>, today)      1     0.007874
863                (</s>, <s>, one)      1     0.007874

[864 rows x 3 columns]
```

Final Task

Subtask:

Summarize the N-gram models built and the insights gained from their counts and conditional probabilities.

Summary:

Data Analysis Key Findings

- **Unigram Model**: A total of 304 unique words were identified from the processed corpus, with 1098 total word occurrences. The probabilities were calculated for each word, summing to 1.0. The tokens `</s>` and `<s>` had the highest probabilities, followed by 'the'.
- **Bigram Model**: The analysis identified 673 unique bigrams out of 1097 total bigram occurrences. Bigram probabilities were calculated as the conditional probability of the second word given the first. Several bigrams exhibited a probability of 1.0, indicating certain word pairs consistently appeared together.
- **Trigram Model**: There were 864 unique trigrams found, comprising 1096 total trigram occurrences. Trigram probabilities were calculated based on the conditional probability of the third word given the preceding two words. Similar to bigrams, numerous trigrams also showed a probability of 1.0, suggesting very specific three-word sequences in the corpus.
- All N-gram models (Unigram, Bigram, and Trigram) were successfully constructed and presented as pandas DataFrames, including their counts and calculated probabilities.

Insights or Next Steps

- The prevalence of N-grams with probabilities of 1.0 suggests a highly structured or repetitive nature within the processed corpus, which could be indicative of specific discourse patterns or data limitations.
- Further analysis could involve smoothing techniques (e.g., Laplace smoothing) for N-grams with zero or very low counts to improve model robustness, especially for predicting unseen sequences.

LETS APPLY SMOOTHING

Apply Smoothing

Why Smoothing is Needed:

N-gram models often suffer from the 'zero-frequency problem,' where N-grams not seen in the training data are assigned a probability of zero. This is problematic because any sequence containing such an N-gram will also have a zero probability, even if other parts of the sequence are common. Smoothing techniques, like Add-one (Laplace) smoothing, address this by reallocating a small portion of probability from seen N-grams to unseen ones, ensuring that all possible N-grams have a non-zero probability. This improves the model's ability to generalize to new, unseen text.

STEP 7

Comparison and Analysis of N-gram Models based on Perplexity

1. **Which model gave the lowest perplexity?** The Trigram model consistently yielded the lowest perplexity values across all evaluated sentences. This indicates its superior performance in predicting sequences compared to the Bigram and Unigram models.

2. **Did trigrams always perform best?** In our specific set of test sentences and training corpus, the Trigram model indeed always performed best, achieving the lowest perplexity for every sentence. This is attributable to its ability to capture richer contextual dependencies by considering two preceding words.

3. **What happens when unseen words appear?** When unseen words (or N-grams) appear, Laplace (Add-1) smoothing ensures that they are assigned a non-zero probability. Without smoothing, such unseen elements would result in a zero probability for any sequence containing them, leading to an infinite perplexity, which makes the model unusable for sequences not explicitly in the training data.

4. **How did smoothing affect results?** Laplace smoothing profoundly affected the results by preventing the 'zero-frequency problem'. It reallocated a small portion of probability mass from observed N-grams to unobserved ones, thus providing a baseline probability for any possible N-gram. This crucial step allowed for meaningful perplexity calculations for all sentences, even those containing N-grams not present in the training data, ensuring model robustness and generalization capabilities.

Overall, the analysis clearly demonstrates that incorporating more context through higher-order N-gram models leads to significantly lower perplexity, making the Trigram model the most effective in predicting sentence likelihood within this dataset. Smoothing further enhances these models by enabling them to handle unseen linguistic phenomena gracefully.

Task

Select at least 5 sentences from the `corpus_text` or create new ones. For each selected sentence, preprocess it using the `preprocess_sentence` function with `remove_stopwords=False`, ensuring start and end tokens are included. Display the original and preprocessed versions of these sentences.

Select and Preprocess Sentences

Subtask:

Select at least 5 sentences from the corpus or create new ones, then preprocess each using the `preprocess_sentence` function with `remove_stopwords=False`, ensuring start and end tokens are included. Display the original and preprocessed versions of these sentences.

Reasoning: To fulfill the subtask, I need to select sentences from the corpus, preprocess them using the `preprocess_sentence` function, and display both the original and preprocessed versions.

```
import nltk

# Ensure punkt is downloaded for sentence tokenization
nltk.download('punkt')

# 1. Select at least 5 sentences from the corpus_text
# Using nltk.sent_tokenize to get sentences from the original corpus
text
all_sentences = nltk.sent_tokenize(corpus_text)

# Select the first 5 sentences for demonstration, or create new ones
if needed.
# Make sure there are at least 5 sentences available in all_sentences.
selected_sentences = all_sentences[:5] if len(all_sentences) >= 5 else
all_sentences

# If the corpus has less than 5 sentences, or for more variety, add
some custom sentences:
if len(selected_sentences) < 5:
    selected_sentences.extend([
        "The students are excited about the upcoming projects.",
        "Learning machine learning can be challenging but rewarding.",
        "Python is a popular language for data science.",
        "They discussed the new assignment with their friends.",
        "Coding helps in problem-solving skills."
    ])
    # Ensure we only take up to 5, or more if the original corpus was
very small
    selected_sentences = selected_sentences[:5]


print("--- Original and Preprocessed Sentences (without Stopword
Removal) ---")
for i, sentence in enumerate(selected_sentences):
    # 3. Print the original sentence
    print(f"\nOriginal Sentence {i+1}: {sentence}")

    # 4. Call preprocess_sentence with remove_stopwords=False
    preprocessed_tokens = preprocess_sentence(sentence,
remove_stopwords=False)

    # 5. Print the resulting preprocessed list of tokens
    print(f"Preprocessed Tokens {i+1}: {preprocessed_tokens}")
```

```
--- Original and Preprocessed Sentences (without Stopword Removal) ---

Original Sentence 1: It was the first day of the semester at Sunrise
Institute of Technology.
Preprocessed Tokens 1: ['<s>', 'it', 'was', 'the', 'first', 'day',
'of', 'the', 'semester', 'at', 'sunrise', 'institute', 'of',
'technology', '</s>']

Original Sentence 2: The campus looked fresh after the rain.
Preprocessed Tokens 2: ['<s>', 'the', 'campus', 'looked', 'fresh',
'after', 'the', 'rain', '</s>']

Original Sentence 3: The trees were green.
Preprocessed Tokens 3: ['<s>', 'the', 'trees', 'were', 'green',
'</s>']

Original Sentence 4: The sky was clear.
Preprocessed Tokens 4: ['<s>', 'the', 'sky', 'was', 'clear', '</s>']

Original Sentence 5: The students were excited.
Preprocessed Tokens 5: ['<s>', 'the', 'students', 'were', 'excited',
'</s>']

[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
```

Reasoning: Since the previous step successfully preprocessed sentences, the next logical step in applying smoothing is to implement Laplace (Add-1) smoothing for the unigram model to address the zero-frequency problem and obtain smoothed probabilities.

```
print("## Applying Laplace (Add-1) Smoothing to Unigram Model")

# Get vocabulary size (V) and total number of words (N)
vocabulary_size_unigram = len(unigram_df) # Number of unique words
total_words_unigram = unigram_df['Count'].sum() # Total occurrences of
words

# Explanation for Laplace Smoothing for Unigrams:
# P_smoothed(word) = (Count(word) + 1) / (Total Words + Vocabulary
Size)
# This method adds 1 to every word's count and adds the vocabulary
size to the total word count.
# This ensures no word has a zero probability and reallocates a small
amount of probability
# to unseen words, making the model more robust.

# Calculate smoothed probabilities
unigram_df['Smoothed_Probability'] = (unigram_df['Count'] + 1) /
(total_words_unigram + vocabulary_size_unigram)

# Add an entry for unseen words to illustrate how they would be
handled
# An unseen word would have a count of 0, so its smoothed probability
would be 1 / (Total Words + Vocabulary Size)
unseen_word_prob = 1 / (total_words_unigram + vocabulary_size_unigram)
print(f"\nProbability for an unseen word (Laplace smoothed):
{unseen_word_prob:.8f}")

print("\nUnigram Model with Laplace Smoothed Probabilities (Top 10):")
print(unigram_df.sort_values(by='Smoothed_Probability',
ascending=False).head(10))
print(f"\nSum of smoothed unigram probabilities:
{unigram_df['Smoothed_Probability'].sum():.4f}")
```

```
## Applying Laplace (Add-1) Smoothing to Unigram Model

Probability for an unseen word (Laplace smoothed): 0.00071327

Unigram Model with Laplace Smoothed Probabilities (Top 10):
    Word  Count  Probability  Smoothed_Probability
0   </s>    128     0.116576              0.092011
1    <s>    128     0.116576              0.092011
2    the     96     0.087432              0.069187
3  arjun     29     0.026412              0.021398
4   they     23     0.020947              0.017118
5   said     22     0.020036              0.016405
6  meera     21     0.019126              0.015692
7  model     20     0.018215              0.014979
8      a     14     0.012750              0.010699
9     is     12     0.010929              0.009272
```

```
Sum of smoothed unigram probabilities: 1.0000
```

Calculate Unigram Sentence Probabilities

Subtask:

For each preprocessed sentence, calculate its probability using the Unigram model. This will be the product of the smoothed probabilities of each word in the sentence, looking up probabilities from `unigram_df`.

Reasoning: To calculate the unigram probability for each preprocessed sentence, I will iterate through the selected sentences, preprocess them to get their tokens, and then multiply the smoothed unigram probabilities of each token. I will use the 'Smoothed_Probability' from 'unigram_df' and handle unseen words using 'unseen_word_prob'.

```
unigram_sentence_probabilities = []

print("\n--- Unigram Sentence Probabilities (Laplace Smoothed) ---")

# Iterate through each selected sentence
for i, original_sentence in enumerate(selected_sentences):
    # Preprocess the sentence to get tokens
    current_preprocessed_tokens = 
preprocess_sentence(original_sentence, remove_stopwords=False)

    # Initialize sentence probability
    sentence_probability = 1.0

    # Calculate the product of smoothed probabilities for each word in 
the sentence
    for word in current_preprocessed_tokens:
        # Look up the smoothed probability from unigram_df
        # Use unseen_word_prob if the word is not in our vocabulary 
(unigram_df)
        word_prob = unigram_df[unigram_df['Word'] == word]
['Smoothed_Probability'].values
        if len(word_prob) > 0:
            sentence_probability *= word_prob[0]
        else:
            # Handle words not in the training vocabulary by using the 
unseen_word_prob
            sentence_probability *= unseen_word_prob

    unigram_sentence_probabilities.append(sentence_probability)

    print(f"\nOriginal Sentence {i+1}: {original_sentence}")
    print(f"Preprocessed Tokens: {current_preprocessed_tokens}")
    print(f"Unigram Probability: {sentence_probability:.10f}")
```

```
--- Unigram Sentence Probabilities (Laplace Smoothed) ---

Original Sentence 1: It was the first day of the semester at Sunrise
Institute of Technology.
Preprocessed Tokens: ['<s>', 'it', 'was', 'the', 'first', 'day', 'of',
'the', 'semester', 'at', 'sunrise', 'institute', 'of', 'technology',
'</s>']
Unigram Probability: 0.0000000000

Original Sentence 2: The campus looked fresh after the rain.
Preprocessed Tokens: ['<s>', 'the', 'campus', 'looked', 'fresh',
'after', 'the', 'rain', '</s>']
Unigram Probability: 0.0000000000

Original Sentence 3: The trees were green.
Preprocessed Tokens: ['<s>', 'the', 'trees', 'were', 'green', '</s>']
Unigram Probability: 0.0000000000

Original Sentence 4: The sky was clear.
Preprocessed Tokens: ['<s>', 'the', 'sky', 'was', 'clear', '</s>']
Unigram Probability: 0.0000000000

Original Sentence 5: The students were excited.
Preprocessed Tokens: ['<s>', 'the', 'students', 'were', 'excited',
'</s>']
Unigram Probability: 0.0000000000
```

Reasoning: The previous calculation resulted in probabilities of 0.0 due to floating-point underflow when multiplying many small numbers. To get meaningful results, I will recalculate the unigram sentence probabilities using logarithms, summing the log probabilities and then exponentiating the result. This will prevent underflow and accurately represent very small probabilities.

```python
import numpy as np # Ensure numpy is imported for log and exp
functions

unigram_sentence_log_probabilities = []

print("\n--- Unigram Sentence Probabilities (Laplace Smoothed, Log
Scale) ---")

# Iterate through each selected sentence
for i, original_sentence in enumerate(selected_sentences):
    current_preprocessed_tokens =
preprocess_sentence(original_sentence, remove_stopwords=False)

    # Initialize sentence log probability to 0.0 (log(1.0) = 0.0)
    sentence_log_probability = 0.0
    # Calculate the sum of log smoothed probabilities for each word in
the sentence
    for word in current_preprocessed_tokens:
        word_prob = unigram_df[unigram_df['Word'] == word]
['Smoothed_Probability'].values
        if len(word_prob) > 0:
            # Add log probability if word is in vocabulary
            sentence_log_probability += np.log(word_prob[0])
        else:
            # Add log probability for unseen words
            sentence_log_probability += np.log(unseen_word_prob)


unigram_sentence_log_probabilities.append(sentence_log_probability)

    # Convert back to normal probability for display, if desired
    sentence_probability_normal = np.exp(sentence_log_probability)

    print(f"\nOriginal Sentence {i+1}: {original_sentence}")
    print(f"Preprocessed Tokens: {current_preprocessed_tokens}")
    print(f"Unigram Log Probability: {sentence_log_probability:.10f}")
    print(f"Unigram Probability: {sentence_probability_normal:.20f}")
# Display with more precision
```

```
--- Unigram Sentence Probabilities (Laplace Smoothed, Log Scale) ---

Original Sentence 1: It was the first day of the semester at Sunrise
Institute of Technology.
Preprocessed Tokens: ['<s>', 'it', 'was', 'the', 'first', 'day', 'of',
'the', 'semester', 'at', 'sunrise', 'institute', 'of', 'technology',
'</s>']
Unigram Log Probability: -73.4345565468
Unigram Probability: 0.00000000000000000000

Original Sentence 2: The campus looked fresh after the rain.
Preprocessed Tokens: ['<s>', 'the', 'campus', 'looked', 'fresh',
'after', 'the', 'rain', '</s>']
Unigram Log Probability: -39.5802760738
Unigram Probability: 0.00000000000000000646

Original Sentence 3: The trees were green.
Preprocessed Tokens: ['<s>', 'the', 'trees', 'were', 'green', '</s>']
Unigram Log Probability: -26.1838623448
Unigram Probability: 0.00000000000425101922

Original Sentence 4: The sky was clear.
Preprocessed Tokens: ['<s>', 'the', 'sky', 'was', 'clear', '</s>']
Unigram Log Probability: -25.7783972367
Unigram Probability: 0.00000000000637652883
```

```
Original Sentence 5: The students were excited.
Preprocessed Tokens: ['<s>', 'the', 'students', 'were', 'excited',
'</s>']
Unigram Log Probability: -24.1689593242
Unigram Probability: 0.00000000003188264417
```

Apply Laplace (Add-1) Smoothing to Bigram Model

Subtask:

Apply Laplace (Add-1) smoothing to the bigram counts to obtain smoothed bigram probabilities. Display the top 10 smoothed bigrams.

Reasoning: To apply Laplace (Add-1) smoothing to the bigram model, I need to calculate the smoothed probability for each bigram. This involves adding 1 to the bigram count and adding the vocabulary size to the count of the first word (context) in the bigram. I will then update the bigram DataFrame with these smoothed probabilities.

```python
print("## Applying Laplace (Add-1) Smoothing to Bigram Model")

# Get vocabulary size (V) from the unigram model
vocabulary_size_bigram = len(unigram_df) # Using the size of the
unigram vocabulary (V)

# Merge bigram_df with unigram_df to get the counts of the first word
(w1) for the denominator
# Renaming columns to avoid conflicts
bigram_counts_merged = pd.merge(
    bigram_df.rename(columns={'Count': 'Bigram_Count'}),
    unigram_df.rename(columns={'Count': 'Unigram_Count'}),
    left_on=bigram_df['Bigram'].apply(lambda x: x[0]), # Join on the
first word of the bigram
    right_on='Word',
    how='left'
)

# Explanation for Laplace Smoothing for Bigrams:
# P_smoothed(w2 | w1) = (Count(w1, w2) + 1) / (Count(w1) + Vocabulary
Size)
# This method adds 1 to every bigram count and adds the vocabulary
size to the count of the first word (context).
# This ensures no bigram has a zero probability and reallocates a
small amount of probability
# to unseen bigrams, making the model more robust.

# Calculate smoothed probabilities
bigram_counts_merged['Smoothed_Probability'] = \
    (bigram_counts_merged['Bigram_Count'] + 1) / \
    (bigram_counts_merged['Unigram_Count'] + vocabulary_size_bigram)

# Prepare the final smoothed bigram DataFrame
smoothed_bigram_df = bigram_counts_merged[['Bigram', 'Bigram_Count',
'Smoothed_Probability']]
smoothed_bigram_df.columns = ['Bigram', 'Count',
'Smoothed_Probability']

# Sort by smoothed probability in descending order for better
inspection
smoothed_bigram_df =
smoothed_bigram_df.sort_values(by='Smoothed_Probability',
ascending=False).reset_index(drop=True)

print("\nBigram Model with Laplace Smoothed Probabilities (Top 10):")
print(smoothed_bigram_df.head(10))

# Calculate the probability for an unseen bigram (where Bigram_Count =
0 and Unigram_Count for w1 is known)
# For an unseen bigram (w1, w_unseen), assume Count(w1, w_unseen) = 0.
# We can't easily get a single 'unseen_bigram_prob' without knowing
w1.
# However, the minimum non-zero smoothed probability occurs when
Bigram_Count is 0,
# so it would be 1 / (Count(w1) + Vocabulary Size).
# Let's consider an unseen bigram with a common first word, e.g.,
'the'.
# Count('the') = 96 (from unigram_df)
# P_smoothed(unseen | the) = (0 + 1) / (96 + 304) = 1 / 400 = 0.0025
# We can find the smallest smoothed probability to show the effect of
smoothing for very low count bigrams.
min_smoothed_prob = smoothed_bigram_df['Smoothed_Probability'].min()
print(f"\nSmallest smoothed bigram probability observed:
{min_smoothed_prob:.8f}")

## Applying Laplace (Add-1) Smoothing to Bigram Model

Bigram Model with Laplace Smoothed Probabilities (Top 10):
              Bigram  Count  Smoothed_Probability
0         (</s>, <s>)    127              0.296296
1          (<s>, the)     44              0.104167
2        (said, </s>)     17              0.055215
3         (<s>, they)     19              0.046296
4        (the, model)     12              0.032500
5    (the, professor)     10              0.027500
6       (arjun, said)      8              0.027027
7     (replied, </s>)      7              0.025641
8   (continued, </s>)      6              0.022508
9     (the, students)      8              0.022500
```

```
Smallest smoothed bigram probability observed: 0.00462963
```

Calculate Bigram Sentence Probabilities

Subtask:

For each preprocessed sentence, calculate its probability using the Bigram model with Laplace (Add-1) smoothing. This will involve multiplying the conditional smoothed probabilities of each word given its preceding word in the sentence.

Instructions:

1. Initialize an empty list to store the bigram probabilities for each sentence.
2. Iterate through each `preprocessed_tokens` list (from `selected_sentences`).
3. For each preprocessed sentence, initialize a `sentence_log_probability` to 0.0.
4. For each bigram `(w_i-1, w_i)` in the sentence, calculate its smoothed conditional probability:
   - `P_smoothed(w_i | w_i-1) = (Count(w_i-1, w_i) + 1) / (Count(w_i-1) + Vocabulary Size)`.
   - Look up `Count(w_i-1, w_i)` from `bigram_df` (or assume 0 if unseen).
   - Look up `Count(w_i-1)` from `unigram_df` (or assume 0 if unseen).
   - The `Vocabulary Size` is `len(unigram_df)`.
   - Add the logarithm of this smoothed probability to `sentence_log_probability`.
5. Handle the first word of each sentence (`<s>`). Its probability is typically taken from the unigram model, or simply `log(P(<s>))` if `<s>` is handled as a regular word in a sequence. For consistency with the definition `P(w_i | w_i-1)`, consider `P(<s>)` as `P(<s> | START_OF_SEQUENCE)`. For simplicity in this step, we can use the smoothed unigram probability for `<s>` or assume it's part of the bigram calculation `P(word_1 | <s>)`.
6. After iterating through all bigrams in a sentence, exponentiate `sentence_log_probability` to get the final `sentence_probability`.
7. Print the original sentence, its preprocessed tokens, and the calculated bigram log probability and normal probability.

Reasoning: To calculate the bigram sentence probabilities with Laplace smoothing, I will iterate through each preprocessed sentence, extract bigrams, and compute the product of their smoothed conditional probabilities using logarithms to avoid underflow. I will need to look up bigram counts and unigram counts from the respective DataFrames and use the previously determined vocabulary size for the smoothing formula.

```
bigram_sentence_log_probabilities = []

print("\n--- Bigram Sentence Probabilities (Laplace Smoothed, Log
Scale) ---")
```

```
# Get vocabulary size (V) from the unigram model, which is used for
bigram smoothing
vocabulary_size = len(unigram_df)

# Helper function to get smoothed bigram probability
def get_smoothed_bigram_prob(prev_word, current_word):
    # Get count of the bigram (w_i-1, w_i)
    bigram_count_row = bigram_df[bigram_df['Bigram'] == (prev_word,
current_word)]
    bigram_count = bigram_count_row['Count'].values[0] if
len(bigram_count_row) > 0 else 0

    # Get count of the first word (w_i-1)
    unigram_count_row = unigram_df[unigram_df['Word'] == prev_word]
    unigram_count = unigram_count_row['Count'].values[0] if
len(unigram_count_row) > 0 else 0

    # Calculate smoothed probability: P(w_i | w_i-1) = (Count(w_i-1,
w_i) + 1) / (Count(w_i-1) + Vocabulary Size)
    smoothed_prob = (bigram_count + 1) / (unigram_count +
vocabulary_size)
    return smoothed_prob


# Iterate through each selected sentence
for i, original_sentence in enumerate(selected_sentences):
    current_preprocessed_tokens =
preprocess_sentence(original_sentence, remove_stopwords=False)

    # Initialize sentence log probability to 0.0
    sentence_log_probability = 0.0

    # Calculate the sum of log smoothed probabilities for each bigram
in the sentence
    # The loop starts from the second token (index 1) to form bigrams
(w_i-1, w_i)
    for j in range(1, len(current_preprocessed_tokens)):
        prev_word = current_preprocessed_tokens[j-1]
        current_word = current_preprocessed_tokens[j]

        bigram_prob = get_smoothed_bigram_prob(prev_word,
current_word)
        sentence_log_probability += np.log(bigram_prob)

    bigram_sentence_log_probabilities.append(sentence_log_probability)

    # Convert back to normal probability for display
    sentence_probability_normal = np.exp(sentence_log_probability)

    print(f"\nOriginal Sentence {i+1}: {original_sentence}")
    print(f"Preprocessed Tokens: {current_preprocessed_tokens}")
    print(f"Bigram Log Probability: {sentence_log_probability:.10f}")
    print(f"Bigram Probability: {sentence_probability_normal:.20f}") #
Display with more precision
```

```
--- Bigram Sentence Probabilities (Laplace Smoothed, Log Scale) ---

Original Sentence 1: It was the first day of the semester at Sunrise
Institute of Technology.
Preprocessed Tokens: ['<s>', 'it', 'was', 'the', 'first', 'day', 'of',
'the', 'semester', 'at', 'sunrise', 'institute', 'of', 'technology',
'</s>']
Bigram Log Probability: -69.4812185411
Bigram Probability: 0.00000000000000000000

Original Sentence 2: The campus looked fresh after the rain.
Preprocessed Tokens: ['<s>', 'the', 'campus', 'looked', 'fresh',
'after', 'the', 'rain', '</s>']
Bigram Log Probability: -36.1205466038
Bigram Probability: 0.00000000000000020561

Original Sentence 3: The trees were green.
Preprocessed Tokens: ['<s>', 'the', 'trees', 'were', 'green', '</s>']
Bigram Log Probability: -22.6513622595
Bigram Probability: 0.00000000014542476246

Original Sentence 4: The sky was clear.
Preprocessed Tokens: ['<s>', 'the', 'sky', 'was', 'clear', '</s>']
Bigram Log Probability: -22.2491704768
Bigram Probability: 0.00000000021742427721

Original Sentence 5: The students were excited.
Preprocessed Tokens: ['<s>', 'the', 'students', 'were', 'excited',
'</s>']
Bigram Log Probability: -21.1764496020
Bigram Probability: 0.00000000063560132156
```

Apply Laplace (Add-1) Smoothing to Trigram Model

Subtask:

Apply Laplace (Add-1) smoothing to the trigram counts to obtain smoothed trigram probabilities. Display the top 10 smoothed trigrams.

Reasoning: To apply Laplace (Add-1) smoothing to the trigram model, I need to calculate the smoothed probability for each trigram. This involves adding 1 to the trigram count and adding the vocabulary size to the count of the first two words (context) in the trigram. I will then create a new DataFrame with these smoothed probabilities and display the top 10.

```
print("## Applying Laplace (Add-1) Smoothing to Trigram Model")

# Get vocabulary size (V) from the unigram model, which is used for
trigram smoothing
vocabulary_size_trigram = len(unigram_df)

# Merge trigram_df with bigram_df to get the counts of the first two
words (w1, w2) for the denominator
# Renaming columns to avoid conflicts and joining on the (w1, w2) part
of the trigram
trigram_counts_merged = pd.merge(
    trigram_df.rename(columns={'Count': 'Trigram_Count'}),
    bigram_df.rename(columns={'Count': 'Bigram_Context_Count'}),
    left_on=trigram_df['Trigram'].apply(lambda x: (x[0], x[1])), #
Join on the first two words of the trigram
    right_on='Bigram',
    how='left'
)

# Explanation for Laplace Smoothing for Trigrams:
# P_smoothed(w3 | w1, w2) = (Count(w1, w2, w3) + 1) / (Count(w1, w2) +
Vocabulary Size)
# This method adds 1 to every trigram count and adds the vocabulary
size to the count of the preceding bigram (context).
# This ensures no trigram has a zero probability and reallocates a
small amount of probability
# to unseen trigrams, making the model more robust.

# Calculate smoothed probabilities
trigram_counts_merged['Smoothed_Probability'] = \
    (trigram_counts_merged['Trigram_Count'] + 1) / \
    (trigram_counts_merged['Bigram_Context_Count'] +
vocabulary_size_trigram)

# Prepare the final smoothed trigram DataFrame
smoothed_trigram_df = trigram_counts_merged[['Trigram',
'Trigram_Count', 'Smoothed_Probability']]
smoothed_trigram_df.columns = ['Trigram', 'Count',
'Smoothed_Probability']

# Sort by smoothed probability in descending order for better
inspection
smoothed_trigram_df =
smoothed_trigram_df.sort_values(by='Smoothed_Probability',
ascending=False).reset_index(drop=True)

print("\nTrigram Model with Laplace Smoothed Probabilities (Top 10):")
print(smoothed_trigram_df.head(10))

# Calculate the probability for an unseen trigram (where Trigram_Count
= 0 and Bigram_Context_Count for (w1,w2) is known)
# Let's consider an unseen trigram with a common bigram, e.g., ('the',
'students').
# Count('the', 'students') = 8 (from bigram_df)
# P_smoothed(unseen | the, students) = (0 + 1) / (8 + 304) = 1 / 312 =
0.003205
min_smoothed_trigram_prob =
smoothed_trigram_df['Smoothed_Probability'].min()
print(f"\nSmallest smoothed trigram probability observed:
{min_smoothed_trigram_prob:.8f}")

## Applying Laplace (Add-1) Smoothing to Trigram Model

Trigram Model with Laplace Smoothed Probabilities (Top 10):
                    Trigram  Count  Smoothed_Probability
0         (</s>, <s>, the)     44              0.104408
1        (said, </s>, <s>)     17              0.056075
2        (</s>, <s>, they)     19              0.046404
3     (<s>, the, students)      8              0.025862
4     (replied, </s>, <s>)      7              0.025723
5       (asked, </s>, <s>)      6              0.022581
6   (continued, </s>, <s>)      6              0.022581
7      (arjun, said, </s>)      6              0.022436
8      (meera, said, </s>)      5              0.019417
9   (carefully, </s>, <s>)      5              0.019417

Smallest smoothed trigram probability observed: 0.00464037
```

Calculate Trigram Sentence Probabilities

Subtask:

For each preprocessed sentence, calculate its probability using the Trigram model with Laplace (Add-1) smoothing. This will involve multiplying the conditional smoothed probabilities of each word given its preceding two words in the sentence.

Instructions:

1. Initialize an empty list to store the trigram probabilities for each sentence.
2. Iterate through each `preprocessed_tokens` list (from `selected_sentences`).
3. For each preprocessed sentence, initialize a `sentence_log_probability` to 0.0.
4. For each trigram `(w_i-2, w_i-1, w_i)` in the sentence, calculate its smoothed conditional probability:
    - `P_smoothed(w_i | w_i-2, w_i-1) = (Count(w_i-2, w_i-1, w_i) + 1) / (Count(w_i-2, w_i-1) + Vocabulary Size)`.
    - Look up `Count(w_i-2, w_i-1, w_i)` from `trigram_df` (or assume 0 if unseen).
    - Look up `Count(w_i-2, w_i-1)` from `bigram_df` (or assume 0 if unseen).
    - The `Vocabulary Size` is `len(unigram_df)`.

   - Add the logarithm of this smoothed probability to `sentence_log_probability`.
5. Handle the first two words of each sentence (`<s>`, `w1`). Their probabilities are usually calculated using lower-order models (e.g., `P(<s>)` from unigram, `P(w1 | <s>)` from bigram) or handled specially. For this step, the loop for trigrams will start from the third token (index 2), considering `<s>` and `w1` as context for `w2`.
6. After iterating through all trigrams in a sentence, exponentiate `sentence_log_probability` to get the final `sentence_probability`.
7. Print the original sentence, its preprocessed tokens, and the calculated trigram log probability and normal probability.

Reasoning: To calculate the trigram sentence probabilities with Laplace smoothing, I will iterate through each preprocessed sentence. For each sentence, I will extract trigrams and compute the product of their smoothed conditional probabilities using logarithms to avoid underflow. I will need to look up trigram counts and bigram counts from the respective DataFrames and use the previously determined vocabulary size for the smoothing formula. The first two words will serve as context for the first trigram calculation.

```
trigram_sentence_log_probabilities = []

print("\n--- Trigram Sentence Probabilities (Laplace Smoothed, Log
Scale) ---")

# Get vocabulary size (V) from the unigram model, which is used for
trigram smoothing
vocabulary_size = len(unigram_df)

# Helper function to get smoothed trigram probability P(w3 | w1, w2)
def get_smoothed_trigram_prob(w1, w2, w3):
    # Get count of the trigram (w1, w2, w3)
    trigram_count_row = trigram_df[trigram_df['Trigram'] == (w1, w2,
w3)]
    trigram_count = trigram_count_row['Count'].values[0] if
len(trigram_count_row) > 0 else 0

    # Get count of the bigram (w1, w2) which is the context
    bigram_context_count_row = bigram_df[bigram_df['Bigram'] == (w1,
w2)]
    bigram_context_count = bigram_context_count_row['Count'].values[0]
if len(bigram_context_count_row) > 0 else 0

    # Calculate smoothed probability: P(w3 | w1, w2) = (Count(w1, w2,
w3) + 1) / (Count(w1, w2) + Vocabulary Size)
    smoothed_prob = (trigram_count + 1) / (bigram_context_count +
vocabulary_size)
    return smoothed_prob


# Iterate through each selected sentence
for i, original_sentence in enumerate(selected_sentences):
    current_preprocessed_tokens =
preprocess_sentence(original_sentence, remove_stopwords=False)

    # Initialize sentence log probability to 0.0
    sentence_log_probability = 0.0

    # Calculate the sum of log smoothed probabilities for each trigram
in the sentence
    # The loop starts from the third token (index 2) to form trigrams
(w_i-2, w_i-1, w_i)
    # The probability of the first two words will be covered by the
trigram model's first calculation P(w2|<s>,w1)
    # or P(w_i | w_i-2, w_i-1) where w_i-2 and w_i-1 are the start
tokens.
    if len(current_preprocessed_tokens) < 3:
        # Sentences too short for trigrams, assign a default low log
probability or handle as error
        # For simplicity, we'll assign a very small log probability
for now if it's not a valid trigram sequence.
        # This ensures we don't divide by zero or have other issues.
Ideally, we'd fall back to bigram/unigram for these.
        sentence_log_probability = np.log(1e-100) # Very small
probability to prevent errors
    else:
        for j in range(2, len(current_preprocessed_tokens)):
            w1 = current_preprocessed_tokens[j-2]
            w2 = current_preprocessed_tokens[j-1]
            w3 = current_preprocessed_tokens[j]

            trigram_prob = get_smoothed_trigram_prob(w1, w2, w3)
            sentence_log_probability += np.log(trigram_prob)


trigram_sentence_log_probabilities.append(sentence_log_probability)

    # Convert back to normal probability for display
    sentence_probability_normal = np.exp(sentence_log_probability)

    print(f"\nOriginal Sentence {i+1}: {original_sentence}")
    print(f"Preprocessed Tokens: {current_preprocessed_tokens}")
    print(f"Trigram Log Probability: {sentence_log_probability:.10f}")
    print(f"Trigram Probability: {sentence_probability_normal:.20f}")
# Display with more precision
```

```
--- Trigram Sentence Probabilities (Laplace Smoothed, Log Scale) ---

Original Sentence 1: It was the first day of the semester at Sunrise
Institute of Technology.
Preprocessed Tokens: ['<s>', 'it', 'was', 'the', 'first', 'day', 'of',
'the', 'semester', 'at', 'sunrise', 'institute', 'of', 'technology',
'</s>']
Trigram Log Probability: -65.3727158996
Trigram Probability: 0.0000000000000000000

Original Sentence 2: The campus looked fresh after the rain.
Preprocessed Tokens: ['<s>', 'the', 'campus', 'looked', 'fresh',
'after', 'the', 'rain', '</s>']
Trigram Log Probability: -34.6484718459
Trigram Probability: 0.0000000000000089611

Original Sentence 3: The trees were green.
Preprocessed Tokens: ['<s>', 'the', 'trees', 'were', 'green', '</s>']
Trigram Log Probability: -20.2405490874
Trigram Probability: 0.0000000162047084097

Original Sentence 4: The sky was clear.
Preprocessed Tokens: ['<s>', 'the', 'sky', 'was', 'clear', '</s>']
Trigram Log Probability: -20.2438224127
Trigram Probability: 0.0000000161517518463

Original Sentence 5: The students were excited.
Preprocessed Tokens: ['<s>', 'the', 'students', 'were', 'excited',
'</s>']
Trigram Log Probability: -18.7591631018
Trigram Probability: 0.0000000712851355522
```

Display Sentence Probabilities

Subtask:

Present a table or clear output showing each chosen sentence and its calculated probability according to the Unigram, Bigram, and Trigram models. Interpret the results by noting which sentences are more or less likely according to each model.

Reasoning: To display the sentence probabilities from all three models in a clear and organized manner, I will create a pandas DataFrame containing the original sentences and their calculated log probabilities for Unigram, Bigram, and Trigram models.

```python
import pandas as pd

# Create a DataFrame to store the results
sentence_probabilities_df = pd.DataFrame({
    'Original Sentence': selected_sentences,
    'Unigram Log Probability': unigram_sentence_log_probabilities,
    'Bigram Log Probability': bigram_sentence_log_probabilities,
    'Trigram Log Probability': trigram_sentence_log_probabilities
})
```

```
print("Sentence Probabilities Across N-gram Models (Laplace
Smoothed):")
print(sentence_probabilities_df)
```

```
Sentence Probabilities Across N-gram Models (Laplace Smoothed):
                                           Original Sentence  Unigram Log
Probability  \
0  It was the first day of the semester at Sunris...                   -
73.434557
1           The campus looked fresh after the rain.                   -
39.580276
2                                  The trees were green.              -
26.183862
3                                      The sky was clear.             -
25.778397
4                              The students were excited.             -
24.168959

   Bigram Log Probability  Trigram Log Probability
0              -69.481219               -65.372716
1              -36.120547               -34.648472
2              -22.651362               -20.240549
3              -22.249170               -20.243822
4              -21.176450               -18.759163
```

Interpretation of Sentence Probabilities Across N-gram Models

The table above shows the Laplace-smoothed log probabilities for the selected sentences according to the Unigram, Bigram, and Trigram models. Log probabilities are used to prevent underflow when multiplying many small probabilities, allowing for more precise comparison.

Key Observations:

1. **General Trend:** For most sentences, as we move from Unigram to Bigram to Trigram models, the log probabilities generally become *less negative* (i.e., the actual probabilities become *higher*). This indicates that higher-order N-gram models, by incorporating more context, often provide a better fit for sequences seen in the training data or sequences that align well with learned patterns.

2. **Impact of Context:**

   - **Unigram:** The Unigram model treats each word independently. Therefore, the probability of a sentence is simply the product of the probabilities of its individual words. This often results in very low probabilities for longer sentences, as it doesn't account for how words sequence together.
   - **Bigram:** The Bigram model considers the probability of a word given the immediately preceding word. This introduces a basic level of context. We can see for sentences like "The campus looked fresh after the rain." and "The students were excited.", the Bigram log probabilities are significantly less negative than

    Unigram, indicating that the word pairs within these sentences are relatively common in the training corpus.
  - **Trigram:** The Trigram model incorporates even more context, considering the probability of a word given the two preceding words. For all sentences, the Trigram log probabilities are the least negative (or highest probability) among the three models. This suggests that the three-word sequences in these sentences are well-represented and predictable given the training data, benefiting from the richer context.

3. **Sentence Length and Probability:** Longer sentences naturally have lower probabilities (more negative log probabilities) because they involve the multiplication of more individual word or N-gram probabilities, each being less than 1. For example, Sentence 1 is the longest and consistently has the most negative log probabilities across all models.

4. **Relative Likelihood:** By comparing the log probabilities within each model, we can infer relative likelihoods. For instance, Sentence 5 ("The students were excited.") consistently has one of the highest log probabilities (least negative) across all models, implying it is a more 'typical' or 'expected' sentence given the training corpus's patterns, especially when considering the context provided by bigrams and trigrams.

Conclusion:

The increase in log probabilities (less negative values) from Unigram to Bigram to Trigram models demonstrates the value of incorporating context in language modeling. Higher-order N-grams can more accurately capture the grammatical and semantic structures of sentences, making them more adept at predicting plausible sequences. Laplace smoothing, applied here, ensures that even unseen N-grams have a non-zero probability, making the models more robust.

Final Task

Subtask:

Summarize the findings from the sentence probability calculations, comparing the different N-gram models and their interpretations of sentence likelihood.

Summary:

Q&A

The findings from the sentence probability calculations show a clear progression in sentence likelihood across different N-gram models. The log probabilities generally became less negative (indicating higher actual probabilities) as the model incorporated more context, moving from Unigram to Bigram, and then to Trigram models. This indicates that higher-order models, by considering the sequence of words, provide a more accurate and realistic assessment of a sentence's likelihood.

Data Analysis Key Findings

- **Laplace (Add-1) Smoothing Effectiveness**: Laplace smoothing was successfully applied to Unigram, Bigram, and Trigram models, ensuring that all possible N-grams, including unseen ones, were assigned a non-zero probability. For example, an unseen unigram had a smoothed probability of approximately $1/(\text{Total Words} + \text{Vocabulary Size})$.
- **Logarithmic Probabilities for Stability**: Direct multiplication of probabilities for sentences resulted in numerical underflow, yielding 0.0. This issue was resolved by using log probabilities, summing the logarithms of individual N-gram probabilities, which allowed for accurate representation and comparison of very small likelihoods.
- **Unigram Model (No Context)**: The Unigram model treats each word independently, leading to very low (highly negative log) probabilities for sentences, particularly longer ones. For instance, Sentence 1 had a Unigram log probability of approximately -73.43.
- **Bigram Model (Basic Context)**: Incorporating one-word context, the Bigram model significantly increased sentence probabilities (less negative log probabilities) compared to the Unigram model. For example, Sentence 5's log probability improved from approximately -24.16 (Unigram) to -21.17 (Bigram).
- **Trigram Model (Rich Context)**: The Trigram model, considering two-word context, provided the highest probabilities (least negative log probabilities) among the three models for the selected sentences. Sentence 5's log probability further increased to approximately -18.75 (Trigram), indicating a better fit to the training data's sequential patterns.
- **Impact of Sentence Length**: Longer sentences consistently exhibited lower (more negative) log probabilities across all models, as expected, due to the cumulative effect of multiplying more individual N-gram probabilities (each less than 1).
- **Relative Sentence Likelihood**: By comparing log probabilities, it was observed that some sentences were consistently more likely across models. For example, "The students were excited." (Sentence 5) generally had the highest (least negative) log probability, suggesting it is a more "typical" or "expected" sequence within the trained corpus.

Insights or Next Steps

- Higher-order N-gram models provide a more nuanced and contextually aware assessment of sentence likelihood, making them more suitable for tasks requiring an understanding of sequential word patterns, such as text generation or machine translation.
- For further refinement, consider exploring more advanced smoothing techniques beyond Laplace (Add-1), such as Kneser-Ney smoothing, which often yield better performance by distributing probability mass more intelligently.

Task

Perplexity Calculation and Model Comparison

Subtask:

Define a function to calculate perplexity for a given sequence of words (sentence) using a specified N-gram model (Unigram, Bigram, or Trigram) with Laplace smoothing. Then, calculate and display the perplexity for each selected test sentence across all three models. Finally, interpret the results, explaining what perplexity means and why a lower value indicates a better model.

Instructions:

1. **Define `calculate_perplexity` function**:
   - This function should take the preprocessed `sentence_tokens` (list of words including `<s>` and `</s>`), the `model_log_probability` (the log probability of the sentence from the respective N-gram model), and `model_type` (e.g., 'unigram', 'bigram', 'trigram').
   - The perplexity formula is `PP(S) = exp(- (1/N) * log P(S))`.
   - Determine `N`, the number of words in the sequence. A common practice is to count all tokens *except* the initial `<s>`. So `N = len(sentence_tokens) - 1`.
   - Return the calculated perplexity.
2. **Calculate Perplexity for Unigram Model**:
   - Iterate through the `selected_sentences` and their corresponding `unigram_sentence_log_probabilities`.
   - Call the `calculate_perplexity` function for each.
3. **Calculate Perplexity for Bigram Model**:
   - Iterate through the `selected_sentences` and their corresponding `bigram_sentence_log_probabilities`.
   - Call the `calculate_perplexity` function for each.
4. **Calculate Perplexity for Trigram Model**:
   - Iterate through the `selected_sentences` and their corresponding `trigram_sentence_log_probabilities`.
   - Call the `calculate_perplexity` function for each.
5. **Display Perplexity Results**:
   - Create a pandas DataFrame to clearly present the original sentences and their perplexity values for Unigram, Bigram, and Trigram models.
6. **Interpret Perplexity and Compare Models**:
   - Add a text cell explaining what perplexity represents in language modeling (a measure of how well a probability model predicts a sample).
   - Explain why a lower perplexity value indicates a better model (the model is less "perplexed" by the test data, meaning it predicts it with higher probability).
   - Discuss the observed perplexity values from the generated table, comparing the performance of the Unigram, Bigram, and Trigram models. Highlight which model performs best and why (e.g., due to richer context).

- Mention any trends observed with sentence length.

Define Perplexity Calculation Function

Subtask:

Create a function to calculate perplexity for a given sequence of words (sentence) using a specified N-gram model (Unigram, Bigram, or Trigram) with Laplace smoothing. This function will leverage the log probabilities calculated in previous steps.

Reasoning: To address the subtask, I need to define a Python function `calculate_perplexity` that computes perplexity based on the provided sentence tokens, its log probability, and the N-gram model type.

```
import numpy as np

def calculate_perplexity(sentence_tokens, model_log_probability):
    """
    Calculates the perplexity for a given sequence of words using its
log probability.

    Args:
        sentence_tokens (list): A list of preprocessed words for the
sentence, including start and end tokens.
        model_log_probability (float): The log probability of the
sentence calculated by an N-gram model.

    Returns:
        float: The calculated perplexity value.
    """
    # Calculate N, the effective number of words in the sequence.
    # For perplexity, we usually count the actual words being
predicted.
    # If the sentence tokens include '<s>' and '</s>', and we consider
'<s>' as context
    # and '</s>' as a predicted word, then N should be the number of
actual words + '</s>' token.
    # A common approach is to count all tokens except the initial
'<s>'.
    N = len(sentence_tokens) - 1 # Subtract 1 for the initial '<s>'
token

    # Perplexity formula: PP(S) = exp(- (1/N) * log P(S))
    # Handle cases where N might be zero or very small to avoid
errors.
    if N <= 0:
        # For empty or very short sequences, perplexity is undefined
or very high.
        # Assign a large value to indicate undefined or very high
perplexity.
```

```
        return float('inf')

    perplexity = np.exp(- (1 / N) * model_log_probability)
    return perplexity

print("Perplexity calculation function defined.")
```

```
Perplexity calculation function defined.
```

Reasoning: Now that the `calculate_perplexity` function is defined, I will calculate the perplexity for each selected sentence using the Unigram model's log probabilities, as per the instructions.

```
unigram_perplexities = []

print("\n--- Calculating Unigram Perplexities ---")
for i, original_sentence in enumerate(selected_sentences):
    preprocessed_tokens = preprocess_sentence(original_sentence,
remove_stopwords=False)
    log_prob = unigram_sentence_log_probabilities[i]

    perplexity = calculate_perplexity(preprocessed_tokens, log_prob)
    unigram_perplexities.append(perplexity)

    print(f"Original Sentence {i+1}: {original_sentence}")
    print(f"Unigram Perplexity: {perplexity:.2f}")
```

```
--- Calculating Unigram Perplexities ---
Original Sentence 1: It was the first day of the semester at Sunrise
Institute of Technology.
Unigram Perplexity: 189.68
Original Sentence 2: The campus looked fresh after the rain.
Unigram Perplexity: 140.83
Original Sentence 3: The trees were green.
Unigram Perplexity: 188.06
Original Sentence 4: The sky was clear.
Unigram Perplexity: 173.41
Original Sentence 5: The students were excited.
Unigram Perplexity: 125.69
```

Reasoning: Next, I will calculate the perplexity for each selected sentence using the Bigram model's log probabilities, following the instructions.

```
bigram_perplexities = []

print("\n--- Calculating Bigram Perplexities ---")
for i, original_sentence in enumerate(selected_sentences):
    preprocessed_tokens = preprocess_sentence(original_sentence,
remove_stopwords=False)
```

```
    log_prob = bigram_sentence_log_probabilities[i]

    perplexity = calculate_perplexity(preprocessed_tokens, log_prob)
    bigram_perplexities.append(perplexity)

    print(f"Original Sentence {i+1}: {original_sentence}")
    print(f"Bigram Perplexity: {perplexity:.2f}")
```

```
--- Calculating Bigram Perplexities ---
Original Sentence 1: It was the first day of the semester at Sunrise
Institute of Technology.
Bigram Perplexity: 143.01
Original Sentence 2: The campus looked fresh after the rain.
Bigram Perplexity: 91.38
Original Sentence 3: The trees were green.
Bigram Perplexity: 92.78
Original Sentence 4: The sky was clear.
Bigram Perplexity: 85.61
Original Sentence 5: The students were excited.
Bigram Perplexity: 69.08
```

Reasoning: Next, I will calculate the perplexity for each selected sentence using the Trigram model's log probabilities, following the instructions.

```
trigram_perplexities = []

print("\n--- Calculating Trigram Perplexities ---")
for i, original_sentence in enumerate(selected_sentences):
    preprocessed_tokens = preprocess_sentence(original_sentence,
remove_stopwords=False)
    log_prob = trigram_sentence_log_probabilities[i]

    perplexity = calculate_perplexity(preprocessed_tokens, log_prob)
    trigram_perplexities.append(perplexity)

    print(f"Original Sentence {i+1}: {original_sentence}")
    print(f"Trigram Perplexity: {perplexity:.2f}")
```

```
--- Calculating Trigram Perplexities ---
Original Sentence 1: It was the first day of the semester at Sunrise
Institute of Technology.
Trigram Perplexity: 106.64
Original Sentence 2: The campus looked fresh after the rain.
Trigram Perplexity: 76.02
Original Sentence 3: The trees were green.
Trigram Perplexity: 57.29
Original Sentence 4: The sky was clear.
Trigram Perplexity: 57.33
```

```
Original Sentence 5: The students were excited.
Trigram Perplexity: 42.60
```

Reasoning: I will create a pandas DataFrame to display the perplexity results for each sentence across the Unigram, Bigram, and Trigram models as requested in the subtask.

```
import pandas as pd

# Create a DataFrame to store the perplexity results
perplexity_df = pd.DataFrame({
    'Original Sentence': selected_sentences,
    'Unigram Perplexity': unigram_perplexities,
    'Bigram Perplexity': bigram_perplexities,
    'Trigram Perplexity': trigram_perplexities
})

print("\n--- Sentence Perplexity Across N-gram Models (Laplace
Smoothed) ---")
print(perplexity_df.round(2))


--- Sentence Perplexity Across N-gram Models (Laplace Smoothed) ---
                                   Original Sentence  Unigram
Perplexity  \
0  It was the first day of the semester at Sunris...
189.68
1          The campus looked fresh after the rain.
140.83
2                            The trees were green.
188.06
3                               The sky was clear.
173.41
4                       The students were excited.
125.69

   Bigram Perplexity  Trigram Perplexity
0             143.01              106.64
1              91.38               76.02
2              92.78               57.29
3              85.61               57.33
4              69.08               42.60
```

Interpretation of Perplexity Results and Model Comparison

What is Perplexity? Perplexity is a widely used metric in language modeling to evaluate how well a probability distribution or model predicts a sample. In simpler terms, it's a measure of how

Interpretation of Perplexity Results and Model Comparison

What is Perplexity? Perplexity is a widely used metric in language modeling to evaluate how well a probability distribution or model predicts a sample. In simpler terms, it's a measure of how "surprised" the model is by new data. A lower perplexity indicates that the model is better at predicting the next word in a sequence, given the preceding words.

Why a Lower Perplexity is Better: A lower perplexity value means the language model assigns a higher probability to the test data. This implies that the model is more confident and accurate in its predictions. For instance, if a model has a perplexity of 100, it means that, on average, the model is as uncertain about the next word as if it had to choose uniformly from 100 equally likely words. A model with a perplexity of 10 is much better, as it's as uncertain as choosing from only 10 words.

Analysis of Observed Perplexity Values (Laplace Smoothed):

The `perplexity_df` table shows the perplexity for each selected sentence across the Unigram, Bigram, and Trigram models:

```
                                   Original Sentence  Unigram
Perplexity  \
0  It was the first day of the semester at Sunris...
189.68
1             The campus looked fresh after the rain.
140.83
2                               The trees were green.
188.06
3                                  The sky was clear.
173.41
4                          The students were excited.
125.69

   Bigram Perplexity  Trigram Perplexity
0             143.01              106.64
1              91.38               76.02
2              92.78               57.29
3              85.61               57.33
4              69.08               42.60
```

Key Observations:

1. **General Trend:** Across all sentences, there is a clear and consistent trend: **Trigram Perplexity < Bigram Perplexity < Unigram Perplexity**. This indicates that as the N-gram model incorporates more context (moving from individual words in Unigram to two preceding words in Trigram), its ability to predict the test sentences improves significantly.

    - **Unigram Model:** Shows the highest perplexity values (e.g., Sentence 1: 189.68). This is expected because the Unigram model predicts each word independently, without considering any preceding context. It is the most

Interpretation of Perplexity Results and Model Comparison

What is Perplexity? Perplexity is a widely used metric in language modeling to evaluate how well a probability distribution or model predicts a sample. In simpler terms, it's a measure of how "surprised" the model is by new data. A lower perplexity indicates that the model is better at predicting the next word in a sequence, given the preceding words.

Why a Lower Perplexity is Better: A lower perplexity value means the language model assigns a higher probability to the test data. This implies that the model is more confident and accurate in its predictions. For instance, if a model has a perplexity of 100, it means that, on average, the model is as uncertain about the next word as if it had to choose uniformly from 100 equally likely words. A model with a perplexity of 10 is much better, as it's as uncertain as choosing from only 10 words.

Analysis of Observed Perplexity Values (Laplace Smoothed):

The `perplexity_df` table shows the perplexity for each selected sentence across the Unigram, Bigram, and Trigram models:

```
                                   Original Sentence  Unigram
Perplexity  \
0  It was the first day of the semester at Sunris...
189.68
1             The campus looked fresh after the rain.
140.83
2                               The trees were green.
188.06
3                                  The sky was clear.
173.41
4                          The students were excited.
125.69

   Bigram Perplexity  Trigram Perplexity
0             143.01              106.64
1              91.38               76.02
2              92.78               57.29
3              85.61               57.33
4              69.08               42.60
```

Key Observations:

1. **General Trend:** Across all sentences, there is a clear and consistent trend: **Trigram Perplexity < Bigram Perplexity < Unigram Perplexity**. This indicates that as the N-gram model incorporates more context (moving from individual words in Unigram to two preceding words in Trigram), its ability to predict the test sentences improves significantly.
   - **Unigram Model:** Shows the highest perplexity values (e.g., Sentence 1: 189.68). This is expected because the Unigram model predicts each word independently,

without considering any preceding context. It is the most "surprised" by sequences of words.

- **Bigram Model:** Demonstrates substantially lower perplexity values compared to the Unigram model (e.g., Sentence 1: 143.01). This improvement highlights the value of even a single word of context. By considering `P(word_i | word_{i-1})`, the model makes more informed predictions.
- **Trigram Model:** Consistently yields the lowest perplexity values (e.g., Sentence 1: 106.64). The Trigram model, by leveraging two words of context (`P(word_i | word_{i-2}, word_{i-1})`), captures more complex dependencies and sequential patterns, leading to the least

Interpretation of Perplexity Results and Model Comparison

What is Perplexity? Perplexity is a widely used metric in language modeling to evaluate how well a probability distribution or model predicts a sample. In simpler terms, it's a measure of how "surprised" the model is by new data. A lower perplexity indicates that the model is better at predicting the next word in a sequence, given the preceding words.

Why a Lower Perplexity is Better: A lower perplexity value means the language model assigns a higher probability to the test data. This implies that the model is more confident and accurate in its predictions. For instance, if a model has a perplexity of 100, it means that, on average, the model is as uncertain about the next word as if it had to choose uniformly from 100 equally likely words. A model with a perplexity of 10 is much better, as it's as uncertain as choosing from only 10 words.

Analysis of Observed Perplexity Values (Laplace Smoothed):

The `perplexity_df` table shows the perplexity for each selected sentence across the Unigram, Bigram, and Trigram models:

```
                                   Original Sentence  Unigram
Perplexity  \
0  It was the first day of the semester at Sunris...
189.68
1          The campus looked fresh after the rain.
140.83
2                             The trees were green.
188.06
3                                The sky was clear.
173.41
4                         The students were excited.
125.69

   Bigram Perplexity  Trigram Perplexity
0             143.01              106.64
1              91.38               76.02
2              92.78               57.29
3              85.61               57.33
4              69.08               42.60
```

Key Observations:

1. **General Trend:** Across all sentences, there is a clear and consistent trend: **Trigram Perplexity < Bigram Perplexity < Unigram Perplexity**. This indicates that as the N-gram model incorporates more context (moving from individual words in Unigram to two preceding words in Trigram), its ability to predict the test sentences improves significantly.

    - **Unigram Model:** Shows the highest perplexity values (e.g., Sentence 1: 189.68). This is expected because the Unigram model predicts each word independently, without considering any preceding context. It is the most "surprised" by sequences of words.
    - **Bigram Model:** Demonstrates substantially lower perplexity values compared to the Unigram model (e.g., Sentence 1: 143.01). This improvement highlights the value of even a single word of context. By considering `P(word_i | word_{i-1})`, the model makes more informed predictions.
    - **Trigram Model:** Consistently yields the lowest perplexity values (e.g., Sentence 1: 106.64). The Trigram model, by leveraging two words of context (`P(word_i | word_{i-2}, word_{i-1})`), captures more complex dependencies and sequential patterns, leading to the least "surprised" predictions.
2. **Impact of Context:** The reduction in perplexity from Unigram to Bigram and further to Trigram models clearly illustrates the importance of context in language modeling. Models that consider more preceding words are better at predicting the likelihood of a sequence because human language is inherently sequential and context-dependent.

3. **Sentence Length:** Longer sentences generally have higher perplexity (even if the per-word perplexity is similar) because `N` in the perplexity formula is based on the number of words being predicted. However, the *relative* improvement from higher-order models is still evident for all sentence lengths.

Conclusion:

The perplexity analysis confirms that higher-order N-gram models provide a more accurate and robust representation of language, particularly when dealing with sequential data like sentences. The Trigram model, by incorporating the richest context among the three, consistently achieves the lowest perplexity, indicating it is the best predictive model for the given test sentences. Laplace smoothing played a crucial role in ensuring that even unseen N-grams contributed to the probability distribution, making the perplexity values more meaningful and preventing infinite perplexity for sequences not directly observed in the training data.

Interpretation of Perplexity Results and Model Comparison

What is Perplexity? Perplexity is a widely used metric in language modeling to evaluate how well a probability distribution or model predicts a sample. In simpler terms, it's a measure of how "surprised" the model is by new data. A lower perplexity indicates that the model is better at predicting the next word in a sequence, given the preceding words.

Why a Lower Perplexity is Better: A lower perplexity value means the language model assigns a higher probability to the test data. This implies that the model is more confident and accurate in its predictions. For instance, if a model has a perplexity of 100, it means that, on average, the model is as uncertain about the next word as if it had to choose uniformly from 100 equally likely words. A model with a perplexity of 10 is much better, as it's as uncertain as choosing from only 10 words.

Analysis of Observed Perplexity Values (Laplace Smoothed):

The `perplexity_df` table shows the perplexity for each selected sentence across the Unigram, Bigram, and Trigram models:

```
                                   Original Sentence  Unigram
Perplexity  \
0  It was the first day of the semester at Sunris...
189.68
1          The campus looked fresh after the rain.
140.83
2                             The trees were green.
188.06
3                                The sky was clear.
173.41
4                         The students were excited.
125.69

   Bigram Perplexity  Trigram Perplexity
0             143.01              106.64
1              91.38               76.02
2              92.78               57.29
3              85.61               57.33
4              69.08               42.60
```

Key Observations:

1. **General Trend:** Across all sentences, there is a clear and consistent trend: **Trigram Perplexity < Bigram Perplexity < Unigram Perplexity**. This indicates that as the N-gram model incorporates more context (moving from individual words in Unigram to two preceding words in Trigram), its ability to predict the test sentences improves significantly.

   - **Unigram Model:** Shows the highest perplexity values (e.g., Sentence 1: 189.68). This is expected because the Unigram model predicts each word independently, without considering any preceding context. It is the most "surprised" by sequences of words.
   - **Bigram Model:** Demonstrates substantially lower perplexity values compared to the Unigram model (e.g., Sentence 1: 143.01). This improvement highlights the value of even a single word of context. By considering `P(word_i | word_{i-1})`, the model makes more informed predictions.
   - **Trigram Model:** Consistently yields the lowest perplexity values (e.g., Sentence 1: 106.64). The Trigram model, by leveraging two words of context (`P(word_i |`

`word_{i-2}, word_{i-1})`), captures more complex dependencies and sequential patterns, leading to the least "surprised" predictions.

2. **Impact of Context:** The reduction in perplexity from Unigram to Bigram and further to Trigram models clearly illustrates the importance of context in language modeling. Models that consider more preceding words are better at predicting the likelihood of a sequence because human language is inherently sequential and context-dependent.

3. **Sentence Length:** Longer sentences generally have higher perplexity (even if the per-word perplexity is similar) because `N` in the perplexity formula is based on the number of words being predicted. However, the *relative* improvement from higher-order models is still evident for all sentence lengths.

Conclusion:

The perplexity analysis confirms that higher-order N-gram models provide a more accurate and robust representation of language, particularly when dealing with sequential data like sentences. The Trigram model, by incorporating the richest context among the three, consistently achieves the lowest perplexity, indicating it is the best predictive model for the given test sentences. Laplace smoothing played a crucial role in ensuring that even unseen N-grams contributed to the probability distribution, making the perplexity values more meaningful and preventing infinite perplexity for sequences not directly observed in the training data.

Final Task

Subtask:

Provide a comprehensive summary of the perplexity calculations and their implications for the N-gram models, discussing which model performed best and why.

Summary:

Q&A

The Trigram model performed best among the three (Unigram, Bigram, and Trigram). This is because it consistently yielded the lowest perplexity values for all test sentences, indicating that it is the most accurate at predicting the sequences. The Trigram model leverages two preceding words as context, allowing it to capture more complex dependencies and sequential patterns in the language compared to the Bigram (one word context) and Unigram (no context) models.

Data Analysis Key Findings

- A `calculate_perplexity` function was successfully defined, which computes perplexity using the formula $PP(S) = \exp(-(1/N) * \log P(S))$, where N is the number of words excluding the initial `<s>` token. It also handles edge cases by returning `float('inf')` for $N \leq 0$.
- The perplexity values for the selected test sentences consistently showed a clear trend: Trigram Perplexity ¿ Bigram Perplexity ¿ Unigram Perplexity.

    - **Unigram Perplexity:** Ranged from approximately 125.69 to 189.68. For example, the sentence "It was the first day of the semester at Sunris..." had a Unigram Perplexity of 189.68.
    - **Bigram Perplexity:** Showed significant improvement over Unigram, ranging from approximately 69.08 to 143.01. For the same sentence, Bigram Perplexity was 143.01.
    - **Trigram Perplexity:** Consistently the lowest, ranging from approximately 42.60 to 106.64. The example sentence had a Trigram Perplexity of 106.64.
- This trend demonstrates that as the N-gram model incorporates more context (moving from Unigram to Trigram), its ability to predict the test sentences improves, resulting in lower perplexity.
- Laplace smoothing played a crucial role in preventing infinite perplexity for N-grams not present in the training data, ensuring meaningful perplexity calculations.

Insights or Next Steps

- The analysis confirms that incorporating more contextual information (i.e., using higher-order N-gram models) significantly improves a language model's predictive accuracy and reduces its "surprise" when encountering new text.
- For future work, consider exploring even higher-order N-grams (e.g., 4-gram or 5-gram) if the dataset size and computational resources permit, as these might capture even richer linguistic dependencies, potentially leading to further reductions in perplexity. However, be mindful of data sparsity issues that arise with larger N-grams.