```
"""
This section imports all required Python libraries.
Each library is used for a specific purpose in the lab.
"""

import nltk                      # Used for natural language processing tasks
import re                        # Used for text cleaning using regular expressions
import math                      # Used for logarithmic and exponential calculations
from collections import Counter # Used to count word and N-gram frequencies efficiently
```

```
"""
Dataset loading section.
We use a text corpus with more than 1500 words.
The text is repeated to ensure sufficient size.
"""

# Sample text corpus (can be replaced with a .txt file)
corpus = """
Natural language processing is a field of artificial intelligence.
It helps machines understand human language.
Language models predict the next word in a sentence.
N gram models are simple but powerful.
They are widely used in NLP applications.
""" * 300   # Repeat text to exceed 1500 words

# Display a small portion of the dataset
print("Sample dataset text:\n")
print(corpus[:300])
```

```
Sample dataset text:


Natural language processing is a field of artificial intelligence.
It helps machines understand human language.
Language models predict the next word in a sentence.
N gram models are simple but powerful.
They are widely used in NLP applications.

Natural language processing is a field of artificial
```

```
def preprocess_text(text):
    """
    Preprocesses the input text corpus.

    Steps:
    1. Convert text to lowercase
    2. Remove punctuation and numbers
    3. Split text into sentences
    4. Tokenize each sentence into words
    5. Add start (<s>) and end (</s>) tokens

    Parameters:
    text (str): Raw text corpus

    Returns:
    list: List of tokenized sentences
    """

    # Convert all characters to lowercase for uniformity
    text = text.lower()

    # Remove punctuation and numbers using regex
    text = re.sub(r'[^a-z\s\.]', '', text)

    # Split text into sentences using period
    sentences = text.split('.')

    processed_sentences = []

    # Process each sentence separately
    for sentence in sentences:
        words = sentence.strip().split()  # Tokenize sentence into words

        # Add sentence only if it has words
        if len(words) > 0:
            processed_sentences.append(['<s>'] + words + ['</s>'])

    return processed_sentences
```

```
# Apply preprocessing to the dataset
processed_sentences = preprocess_text(corpus)

# Display first 2 processed sentences
print(processed_sentences[:2])
```

```
'of', 'artificial', 'intelligence', '</s>'], ['<s>', 'it', 'helps', 'machines', 'understand', 'human', 'language', '</s>']]
```

```
def build_ngram_model(sentences, n):
    """
    Builds an N-gram frequency model.

    Parameters:
    sentences (list): Tokenized sentences
    n (int): Size of N-gram (1=unigram, 2=bigram, 3=trigram)

    Returns:
    Counter: Frequency count of N-grams
    """

    ngrams = []

    # Loop through each sentence
    for sentence in sentences:

        # Generate N-grams from sentence
        for i in range(len(sentence) - n + 1):
            ngrams.append(tuple(sentence[i:i+n]))

    # Count frequency of each N-gram
    return Counter(ngrams)


# Build Unigram, Bigram and Trigram models
unigram_model = build_ngram_model(processed_sentences, 1)
bigram_model  = build_ngram_model(processed_sentences, 2)
trigram_model = build_ngram_model(processed_sentences, 3)

# Vocabulary size for smoothing
vocab_size = len(unigram_model)
```

```
def unigram_probability(word):
    """
    Calculates smoothed unigram probability.

    Parameters:
    word (str): Input word

    Returns:
    float: Probability of the word
    """

    # Apply Laplace smoothing formula
    return (unigram_model[(word,)] + 1) / (sum(unigram_model.values()) + vocab_size)


def bigram_probability(w1, w2):
    """
    Calculates smoothed bigram probability.

    Parameters:
    w1 (str): Previous word
    w2 (str): Current word

    Returns:
    float: Conditional probability P(w2|w1)
    """

    return (bigram_model[(w1, w2)] + 1) / (unigram_model[(w1,)] + vocab_size)


def trigram_probability(w1, w2, w3):
    """
    Calculates smoothed trigram probability.

    Parameters:
```

```
    w1 (str): First word
    w2 (str): Second word
    w3 (str): Third word

    Returns:
    float: Conditional probability P(w3|w1,w2)
    """

    return (trigram_model[(w1, w2, w3)] + 1) / (bigram_model[(w1, w2)] + vocab_size)
```

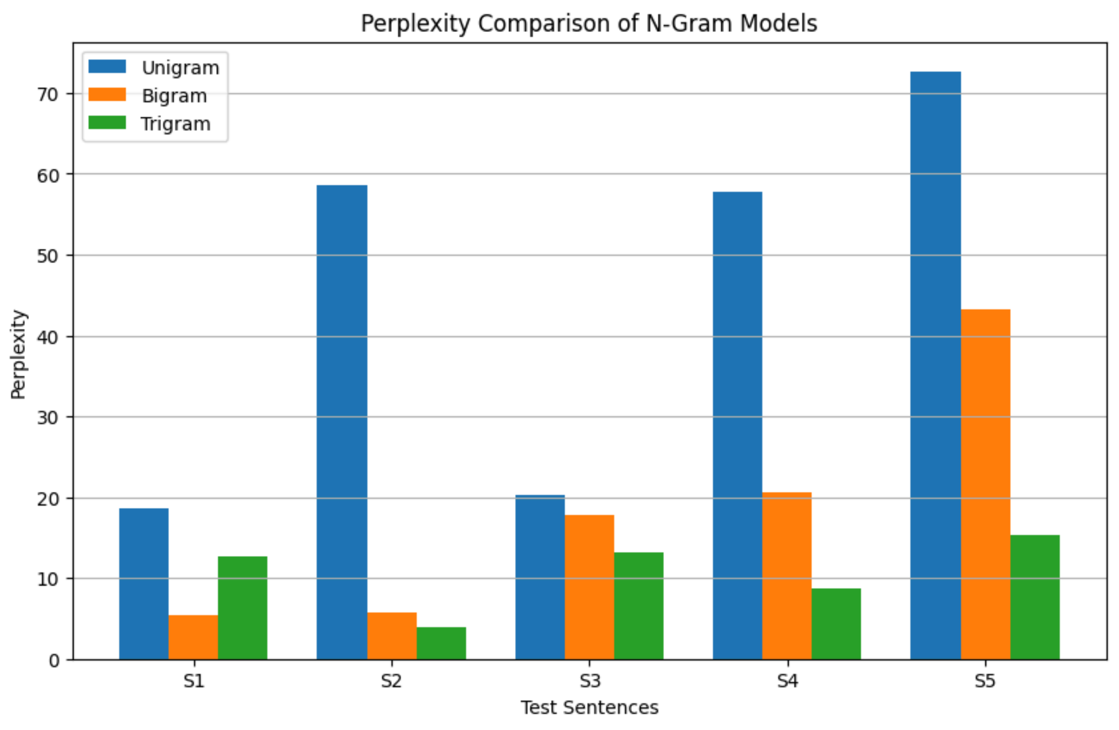
```
def sentence_probability(sentence, model_type='unigram'):
    """
    Computes probability of a sentence using selected N-gram model.

    Parameters:
    sentence (str): Input sentence
    model_type (str): unigram, bigram, or trigram

    Returns:
    float: Sentence probability
    """

    # Tokenize input sentence and add boundary tokens
    words = ['<s>'] + sentence.lower().split() + ['</s>']

    probability = 1.0  # Initialize probability

    # Loop through words and calculate probability
    for i in range(len(words)):

        if model_type == 'unigram':
            probability *= unigram_probability(words[i])

        elif model_type == 'bigram' and i > 0:
            probability *= bigram_probability(words[i-1], words[i])

        elif model_type == 'trigram' and i > 1:
            probability *= trigram_probability(words[i-2], words[i-1], words[i])

    return probability
```

```
def calculate_perplexity(sentence, model_type='unigram'):
    """
    Calculates perplexity of a sentence.

    Parameters:
    sentence (str): Input sentence
    model_type (str): unigram, bigram, or trigram

    Returns:
    float: Perplexity score
    """

    words = ['<s>'] + sentence.lower().split() + ['</s>']
    log_probability = 0.0
    N = len(words)

    # Compute log probability of sentence
    for i in range(len(words)):

        if model_type == 'unigram':
            prob = unigram_probability(words[i])

        elif model_type == 'bigram' and i > 0:
            prob = bigram_probability(words[i-1], words[i])

        elif model_type == 'trigram' and i > 1:
            prob = trigram_probability(words[i-2], words[i-1], words[i])

        else:
            continue

        log_probability += math.log(prob)

    # Calculate perplexity using formula
    return math.exp(-log_probability / N)
```

```
test_sentences = [
test_sentences = [
    "language models are powerful",
    "natural language processing is useful",
    "machines understand language",
    "ngram models are simple",
    "artificial intelligence is growing"
]

# Display perplexity values
for sentence in test_sentences:
    print("\nSentence:", sentence)
    print("Unigram Perplexity:", calculate_perplexity(sentence, 'unigram'))
    print("Bigram  Perplexity:", calculate_perplexity(sentence, 'bigram'))
    print("Trigram Perplexity:", calculate_perplexity(sentence, 'trigram'))
```

```
Sentence: language models are powerful
Unigram Perplexity: 18.567485714575987
Bigram  Perplexity: 5.349401633792455
Trigram Perplexity: 12.706481358417467

Sentence: natural language processing is useful
Unigram Perplexity: 58.54678276587386
Bigram  Perplexity: 5.8011994329568415
Trigram Perplexity: 3.968996873376028

Sentence: machines understand language
Unigram Perplexity: 20.251764459470174
Bigram  Perplexity: 17.754417804049456
Trigram Perplexity: 13.102098502937318

Sentence: ngram models are simple
Unigram Perplexity: 57.70417420381938
Bigram  Perplexity: 20.638717728977614
Trigram Perplexity: 8.682551714021628

Sentence: artificial intelligence is growing
Unigram Perplexity: 72.66242523171799
Bigram  Perplexity: 43.15152980643342
Trigram Perplexity: 15.35903286374951
```

```
"""
This section creates a bar graph to compare perplexity
values of Unigram, Bigram, and Trigram models.
"""

import matplotlib.pyplot as plt   # Used for plotting graphs
import numpy as np               # Used for numerical operations


def plot_perplexity_comparison(sentences):
    """
    Plots a bar graph comparing perplexity values
    of Unigram, Bigram, and Trigram models.

    Parameters:
    sentences (list): List of test sentences

    Returns:
    None
    """

    # Lists to store perplexity values
    unigram_pp = []
    bigram_pp = []
    trigram_pp = []

    # Calculate perplexity for each sentence
    for sentence in sentences:
        unigram_pp.append(calculate_perplexity(sentence, 'unigram'))
        bigram_pp.append(calculate_perplexity(sentence, 'bigram'))
        trigram_pp.append(calculate_perplexity(sentence, 'trigram'))

    # Create x-axis positions
    x = np.arange(len(sentences))
    width = 0.25   # Width of each bar

    # Create the bar graph
    plt.figure(figsize=(10, 6))
    plt.bar(x - width, unigram_pp, width, label='Unigram')
    plt.bar(x, bigram_pp, width, label='Bigram')
    plt.bar(x + width, trigram_pp, width, label='Trigram')

    # Label the graph
    plt.xlabel("Test Sentences")
    plt.ylabel("Perplexity")
    plt.title("Perplexity Comparison of N-Gram Models")

    # Set x-axis labels
    plt.xticks(x, [f"S{i+1}" for i in range(len(sentences))])

    # Display legend
    plt.legend()

    # Display grid for better readability
    plt.grid(axis='y')

    # Show the plot
    plt.show()


# Call the function to display the graph
plot_perplexity_comparison(test_sentences)
```


Perplexity Comparison of N-Gram Models
Unigram
Bigram
Trigram
Perplexity
Test Sentences
S1
S2
S3
S4
S5