1.IMPORT REQUIRED LIBRARIES

Import libraries for:

• text preprocessing

• tokenization

• counting N-grams

• probability calculations

```
import re
import string
import nltk
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer, WordNetLemmatizer
from nltk.tokenize import word_tokenize, sent_tokenize
from nltk.util import ngrams
from collections import Counter
import numpy as np
import pandas as pd
import math
```

2.LOAD DATASET load or copy text corpus

• clean unnecessary lines

• display sample text

• explain dataset in 5–6 lines

```
import nltk
nltk.download('gutenberg')
from nltk.corpus import gutenberg

# Load a sample text (Jane Austen's Emma)
raw_text = gutenberg.raw('austen-emma.txt')
import re

# Remove extra whitespace
clean_text = re.sub(r'\s+', ' ', raw_text)

# Remove non-alphabetic characters (optional)
clean_text = re.sub(r'[^a-zA-Z\s]', '', clean_text)

print(clean_text[:500])  # Display first 500 characters
```

```
Emma by Jane Austen  VOLUME I CHAPTER I Emma Woodhouse handsome clever and rich with a comfortable home and happy dispositio
[nltk_data] Downloading package gutenberg to /root/nltk_data...
[nltk_data]   Package gutenberg is already up-to-date!
```

3.PREPROCESS TEXT Write functions to:

• convert to lowercase

• remove punctuation and numbers

• tokenize words

• optionally remove stopwords

• add start/end tokens for sentences (e.g., ,̅)

```
#convert to lowr case
def to_lowercase(text):
    return text.lower()
# remove punctuation and numbers
def remove_punctuation_numbers(text):
    # Remove punctuation
    text = text.translate(str.maketrans('', '', string.punctuation))

    # Remove numbers
    text = re.sub(r'\d+', '', text)

    return text
```

```
#tokenise the words
def tokenize_words(text):
    return word_tokenize(text)
#removing the stop words
def remove_stopwords(tokens, remove=True):
    if not remove:
        return tokens

    stop_words = set(stopwords.words('english'))
    filtered_tokens = [word for word in tokens if word not in stop_words]

    return filtered_tokens
# add start/end tokens for the sentences
def add_sentence_tokens(text):
    sentences = sent_tokenize(text)

    processed_sentences = []
    for sentence in sentences:
        tokens = word_tokenize(sentence)
        tokens = ['<s>'] + tokens + ['</s>']
        processed_sentences.append(tokens)

    return processed_sentences
```

4.BUILD N-GRAMS MODELS

• construct Unigram

• construct Bigram

• construct Trigram

```
#UNIGRAM MODEL
def build_unigram(tokens):
    """
    tokens: list of words
    returns: dictionary of unigram counts
    """
    return Counter(tokens)
# BIGRAM MODEL
def build_bigram(tokens):
    """
    tokens: list of words
    returns: dictionary of bigram counts
    """
    bigrams = ngrams(tokens, 2)
    return Counter(bigrams)
# 3 TRIGRAM MODELS
def build_trigram(tokens):
    """
    tokens: list of words
    returns: dictionary of trigram counts
    """
    trigrams = ngrams(tokens, 3)
    return Counter(trigrams)
```

COUNT TABLES Create tables showing:

• word counts

• conditional probabilities

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

# PREPROCESS TEXT TO GET TOKENS
processed_text = to_lowercase(clean_text)
processed_text = remove_punctuation_numbers(processed_text)
tokens = tokenize_words(processed_text)
# Optionally remove stopwords if desired
# tokens = remove_stopwords(tokens, remove=True)

# UNIGRAM MODEL
unigram_counts = build_unigram(tokens)
unigram_table = pd.DataFrame(
    unigram_counts.items(),
    columns=["Word", "Count"]
).sort_values(by="Count", ascending=False)
```

```
print("UNIGRAM TABLE:")
print(unigram_table.head())
print("\n")

# BIGRAM MODEL
bigram_counts = build_bigram(tokens)
bigram_probabilities = []

for (w1, w2), count in bigram_counts.items():
    # Ensure w1 exists in unigram_counts to avoid division by zero or KeyError
    if unigram_counts.get(w1, 0) > 0:
        prob = count / unigram_counts[w1]
        bigram_probabilities.append((w1, w2, count, prob))
    else:
        # Handle cases where w1 might not be in unigram_counts (e.g., start tokens, or very rare words)
        bigram_probabilities.append((w1, w2, count, 0.0)) # Assign 0 probability or handle as appropriate

bigram_table = pd.DataFrame(
    bigram_probabilities,
    columns=["Previous Word (w1)", "Next Word (w2)", "Bigram Count", "P(w2|w1)"]
).sort_values(by="Bigram Count", ascending=False)

print("BIGRAM TABLE:")
print(bigram_table.head())
print("\n")

# TRIGRAM MODEL
trigram_counts = build_trigram(tokens)
trigram_table = pd.DataFrame(
    trigram_counts.items(),
    columns=["Trigram", "Count"]
).sort_values(by="Count", ascending=False)

print("TRIGRAM TABLE:")
print(trigram_table.head())
```

```
[nltk_data] Downloading package punkt_tab to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt_tab.zip.
UNIGRAM TABLE:
   Word  Count
19   to   5149
23  the   5146
10  and   4613
22   of   4274
13    a   3073


BIGRAM TABLE:
     Previous Word (w1) Next Word (w2)  Bigram Count  P(w2|w1)
837                  to             be           602  0.116916
25                   of            the           563  0.131727
37                   in            the           442  0.205869
145                  it            was           418  0.174167
1227                  i             am           363  0.122305


TRIGRAM TABLE:
                Trigram  Count
2642       (i, do, not)    123
1391      (i, am, sure)     97
9876  (she, could, not)     68
1039   (a, great, deal)     63
2056    (it, would, be)     61
```

5.APPLY SMOOTHING

Add-one (Laplace) smoothing

```
trigram_laplace = []

# Define V as the vocabulary size
V = len(unigram_counts)

for (w1, w2, w3), count in trigram_counts.items():

    numerator = count + 1
    denominator = bigram_counts.get((w1, w2), 0) + V # Use .get() to handle cases where (w1, w2) might not be in bigram_cou

    # Ensure denominator is not zero before division
    if denominator == 0:
        probability = 0.0
    else:
        probability = numerator / denominator
    trigram_laplace.append((w1, w2, w3, count, probability))

trigram_laplace_table = pd.DataFrame(
    trigram_laplace,
    columns=["w1", "w2", "w3", "Original Count", "Laplace P(w3|w1,w2)"]
)

print(trigram_laplace_table)
def laplace_trigram_prob(w1, w2, w3):
    count_trigram = trigram_counts.get((w1, w2, w3), 0)
    count_bigram = bigram_counts.get((w1, w2), 0)

    return (count_trigram + 1) / (count_bigram + V)
```

```
                w1         w2         w3  Original Count  Laplace P(w3|w1,w2)
0             emma         by       jane               1             0.000215
1               by       jane     austen               1             0.000215
2             jane     austen     volume               1             0.000215
3           austen     volume          i               1             0.000215
4           volume          i    chapter               1             0.000215
...            ...        ...        ...             ...                  ...
130158          in        the    perfect               1             0.000205
130159         the    perfect  happiness               1             0.000215
130160     perfect  happiness         of               1             0.000215
130161          of        the      union               1             0.000203
130162         the      union      finis               1             0.000215

[130163 rows x 5 columns]
```

6.SENTENCE PROBABILITY PREDICTION

• choose at least 5 sentences

• compute probability using:

o Unigram model

o Bigram model

o Trigram model

```
    # Define N (total number of words in the corpus)
    # This should be calculated after unigram_counts is built in a prior cell.
    # The kernel state confirms unigram_counts is available.
    N = sum(unigram_counts.values()) # total number of tokens (words)

    # Helper functions for individual n-gram probabilities with Laplace smoothing
    # (These compute P(word), P(w2|w1), P(w3|w1,w2) respectively)
    def get_unigram_word_prob(word):
        """Calculates P(word) with Laplace smoothing."""
        return (unigram_counts.get(word, 0) + 1) / (N + V)

    def get_bigram_conditional_prob(w1, w2):
        """Calculates P(w2|w1) with Laplace smoothing."""
        # Denominator is Count(w1) + V
        return (bigram_counts.get((w1, w2), 0) + 1) / \
               (unigram_counts.get(w1, 0) + V)

    def get_trigram_conditional_prob(w1, w2, w3):
        """Calculates P(w3|w1, w2) with Laplace smoothing."""
        # Denominator is Count(w1, w2) + V
        return (trigram_counts.get((w1, w2, w3), 0) + 1) / \
               (bigram_counts.get((w1, w2), 0) + V)


    # Functions to calculate sentence probabilities
    def calculate_sentence_unigram_prob(sentence_tokens):
        """Calculates the probability of a sentence using a unigram model with Laplace smoothing."""
        sentence_probability = 1.0
        for word in sentence_tokens:
            sentence_probability *= get_unigram_word_prob(word)
        return sentence_probability

    def calculate_sentence_bigram_prob(sentence_tokens):
        """Calculates the probability of a sentence using a bigram model with Laplace smoothing."""
        sentence_probability = 1.0
        # P(S) = P(w1|<s>) * P(w2|w1) * ... * P(</s>|wn)
        # The loop iterates through (w_{i-1}, w_i) pairs starting from (<s>, w1).
        for i in range(1, len(sentence_tokens)):
            w1 = sentence_tokens[i-1] # previous word
            w2 = sentence_tokens[i]   # current word
            sentence_probability *= get_bigram_conditional_prob(w1, w2)
        return sentence_probability

    def calculate_sentence_trigram_prob(sentence_tokens):
        """Calculates the probability of a sentence using a trigram model with Laplace smoothing."""
        sentence_probability = 1.0

        # Start with the probability of the first bigram (P(w1|<s>))
        # This assumes sentence_tokens[0] is '<s>' and sentence_tokens[1] is the actual first word.
        # This is equivalent to P(token_1 | token_0)
        if len(sentence_tokens) > 1: # Need at least '<s>' and one more word
            sentence_probability *= get_bigram_conditional_prob(sentence_tokens[0], sentence_tokens[1])

        # Then calculate the rest using trigram probabilities
        # P(token_i | token_{i-2}, token_{i-1}) for i >= 2
        for i in range(2, len(sentence_tokens)):
            w1 = sentence_tokens[i-2] # w_{i-2}
            w2 = sentence_tokens[i-1] # w_{i-1}
            w3 = sentence_tokens[i]   # w_i
            sentence_probability *= get_trigram_conditional_prob(w1, w2, w3)
        return sentence_probability


    #SAMPLE SENTENCES
    sentences = [
        ['<s>', 'i', 'love', 'nlp', '</s>'],
        ['<s>', 'machine', 'learning', 'is', 'fun', '</s>'],
        ['<s>', 'nlp', 'is', 'powerful', '</s>'],
        ['<s>', 'i', 'enjoy', 'nlp', '</s>'],
        ['<s>', 'learning', 'is', 'fun', '</s>']
    ]

    for s in sentences:
        print("Sentence:", " ".join(s))
        # Call the new sentence probability functions
        print("Unigram:", calculate_sentence_unigram_prob(s))
        print("Bigram :", calculate_sentence_bigram_prob(s))
        print("Trigram:", calculate_sentence_trigram_prob(s))
        print("-"*50)
```

```
Sentence: <s> i love nlp </s>
Unigram: 2.584049166005266e-21
Bigram : 5.994638027314515e-16
Trigram: 1.3332180555873587e-16
--------------------------------------------------
Sentence: <s> machine learning is fun </s>
Unigram: 5.559674098722205e-29
Bigram : 1.2661931222853335e-20
Trigram: 1.4335673877781828e-20
--------------------------------------------------
Sentence: <s> nlp is powerful </s>
Unigram: 5.589874248426464e-23
Bigram : 1.1775599437342545e-16
Trigram: 1.333934454327599e-16
--------------------------------------------------
Sentence: <s> i enjoy nlp </s>
Unigram: 2.0222993473084694e-22
Bigram : 1.0104781055026203e-16
Trigram: 1.333934454327599e-16
--------------------------------------------------
Sentence: <s> learning is fun </s>
Unigram: 9.316457080710773e-24
Bigram : 1.1781927002865029e-16
Trigram: 1.333934454327599e-16
--------------------------------------------------
```

7.PERPLEXITY CALCULATION

• compute perplexity for test sentences

• compare perplexity across models

```
#UNIGRAM PERPLEXITY

def unigram_perplexity(sentence):
    log_prob = 0
    N = len(sentence)

    for word in sentence:
        log_prob += math.log(unigram_prob(word))

    return math.exp(-log_prob / N)
#BIGRAM PERPLEXITY
```

```
def bigram_perplexity(sentence):
    log_prob = 0
    N = len(sentence) - 1

    for i in range(1, len(sentence)):
        log_prob += math.log(
            bigram_prob(sentence[i-1], sentence[i])
        )

    return math.exp(-log_prob / N)
# TRIGRAM PERPLEXITY
def trigram_perplexity(sentence):
    log_prob = 0
    N = len(sentence) - 2

    for i in range(2, len(sentence)):
        log_prob += math.log(
            trigram_prob(sentence[i-2],
                         sentence[i-1],
                         sentence[i])
        )

    return math.exp(-log_prob / N)
for s in sentences:
    print("Sentence:", " ".join(s))
    print("Unigram Perplexity :", unigram_perplexity(s))
    print("Bigram Perplexity  :", bigram_perplexity(s))
    print("Trigram Perplexity :", trigram_perplexity(s))
    print("-"*50)
```

```
Sentence: <s> i love nlp </s>
Unigram Perplexity : 13108.103005559939
Bigram Perplexity  : 6390.859338424181
Trigram Perplexity : 9306.666368230455
--------------------------------------------------
Sentence: <s> machine learning is fun </s>
Unigram Perplexity : 51186.85471778906
Bigram Perplexity  : 9538.937647228502
Trigram Perplexity : 9304.999999999993
--------------------------------------------------
Sentence: <s> nlp is powerful </s>
Unigram Perplexity : 28217.568786500913
Bigram Perplexity  : 9599.624466957386
Trigram Perplexity : 9304.999999999993
--------------------------------------------------
Sentence: <s> i enjoy nlp </s>
Unigram Perplexity : 21818.80252566707
Bigram Perplexity  : 9973.974947456942
Trigram Perplexity : 9304.999999999993
--------------------------------------------------
Sentence: <s> learning is fun </s>
Unigram Perplexity : 40378.46847744365
Bigram Perplexity  : 9598.335321092436
Trigram Perplexity : 9304.999999999993
--------------------------------------------------
```

8.COMPARISON AND ANALYSIS

1.Which model gave lowest perplexity?

A:Usually, the trigram model gave the lowest perplexity because it looks at two previous words, so it understands context better.

2.Did trigrams always perform best?

A:No, not always. Trigrams work best when the sentence pattern was already seen in training. If the data is small or the sentence is new, trigrams can perform worse than bigrams.

3.What happens when unseen words appear?

A:If a word was never seen in training:

Without smoothing → probability becomes 0 → perplexity becomes infinite.

With smoothing → probability becomes very small, but not zero → model still works.

Unseen words increase perplexity.

4.How did smoothing affect results?

A:Smoothing: Prevented zero probabilities, Made perplexity finite, Made the model more stable, But it slightly reduces probabilities of common words.