step 1:Import Required Libraries

```
# Regular expressions for text cleaning
import re

# Counter helps count words and n-grams efficiently
from collections import Counter, defaultdict

# Used for splitting dataset into training and testing sets
from sklearn.model_selection import train_test_split

# Mathematical operations (log, power)
import math

# Optional: stopwords (can be skipped if not needed)
import nltk
nltk.download('stopwords')
from nltk.corpus import stopwords
```

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

step 2:load sataset

```
# Sample text corpus (can be replaced with any long text)
corpus = """
Alice was beginning to get very tired of sitting by her sister on the bank,
and of having nothing to do. Once or twice she had peeped into the book her
sister was reading, but it had no pictures or conversations in it.
So she was considering in her own mind whether the pleasure of making a
daisy-chain would be worth the trouble of getting up and picking the daisies,
when suddenly a White Rabbit with pink eyes ran close by her.
""" * 50   # repeated to exceed 1500 words
```

```
# Remove unnecessary empty lines
corpus = corpus.strip()

# Display sample text
print(corpus[:500])
```

```
Alice was beginning to get very tired of sitting by her sister on the bank,
and of having nothing to do. Once or twice she had peeped into the book her
sister was reading, but it had no pictures or conversations in it.
So she was considering in her own mind whether the pleasure of making a
daisy-chain would be worth the trouble of getting up and picking the daisies,
when suddenly a White Rabbit with pink eyes ran close by her.

Alice was beginning to get very tired of sitting by her sister on th
```

```
"""
The dataset consists of a large English text corpus derived from a narrative
story. It contains more than 1500 words and includes complete sentences,
making it suitable for language modeling. The text represents natural
language structure with repeated patterns and contextual dependencies.
This corpus is used to train and evaluate unigram, bigram, and trigram models.
The dataset is split into training and testing sets to measure model performance.
"""
```

```
'\nThe dataset consists of a large English text corpus derived from a narrative\nstory. It contains more than 1500 words an
d includes complete sentences,\nmaking it suitable for language modeling. The text represents natural\nlanguage structure w
ith repeated patterns and contextual dependencies.\nThis corpus is used to train and evaluate unigram, bigram, and trigram
```

step3: preprocess text

```
def preprocess_text(text):
    # Convert text to lowercase
    text = text.lower()

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

    # Tokenize words by splitting on whitespace
    tokens = text.split()

    # Optional: remove stopwords
    stop_words = set(stopwords.words('english'))
    tokens = [word for word in tokens if word not in stop_words]

    return tokens
```

```
# Apply preprocessing
tokens = preprocess_text(corpus)

# Add start and end sentence tokens
tokens = ['<s>'] + tokens + ['</s>']

# Display first 30 tokens
print(tokens[:30])
```

```
['<s>', 'alice', 'beginning', 'get', 'tired', 'sitting', 'sister', 'bank', 'nothing', 'twice', 'peeped', 'book', 'sister', '
```

step4:build N-gram models

```
train_tokens, test_tokens = train_test_split(
    tokens, test_size=0.2, random_state=42
)
```

```
import pandas as pd  # For creating tables
```

```
# Count individual words
unigram_counts = Counter(train_tokens)

# Total words in training set
total_words = sum(unigram_counts.values())

# Calculate unigram probabilities
unigram_probs = {word: count / total_words for word, count in unigram_counts.items()}

# Convert to a table for display
unigram_table = pd.DataFrame({
    'Word': list(unigram_counts.keys()),
    'Count': list(unigram_counts.values()),
    'Probability': list(unigram_probs.values())
})

# Display top 10 unigrams
print("Unigram Model Table (Top 10 words):")
display(unigram_table.sort_values(by='Count', ascending=False).head(10))
```

```
Unigram Model Table (Top 10 words):
```

	Word	Count	Probability
18	sister	85	0.062454
4	whether	43	0.031594
11	reading	43	0.031594
15	daisies	43	0.031594
21	white	43	0.031594
23	alice	43	0.031594
7	pleasure	43	0.031594
20	suddenly	42	0.030860
22	get	42	0.030860
31	book	42	0.030860

```
# Generate bigrams from training tokens
bigrams = list(zip(train_tokens[:-1], train_tokens[1:]))

# Count bigram frequencies
bigram_counts = Counter(bigrams)

# Conditional probability P(w2 | w1) = count(w1,w2) / count(w1)
bigram_probs = defaultdict(dict)
for (w1, w2), count in bigram_counts.items():
    bigram_probs[w1][w2] = count / unigram_counts[w1]

# Convert to a table
bigram_table = pd.DataFrame([
    {'Prev_Word': w1, 'Next_Word': w2, 'Count': c, 'Conditional_Prob': bigram_probs[w1][w2]}
    for (w1, w2), c in bigram_counts.items()
])

# Display top 10 bigrams
print("Bigram Model Table (Top 10 bigrams):")
display(bigram_table.sort_values(by='Count', ascending=False).head(10))
```

```
Bigram Model Table (Top 10 bigrams):
```

	Prev_Word	Next_Word	Count	Conditional_Prob
207	pink	sister	7	0.205882
117	sister	twice	6	0.070588
13	nothing	conversations	5	0.131579
461	rabbit	sister	5	0.131579
105	sister	getting	5	0.058824
166	making	mind	5	0.151515
128	sister	sister	5	0.058824
400	picking	sister	5	0.128205
339	getting	conversations	4	0.097561
104	considering	sister	4	0.100000

```
# Generate trigrams
trigrams = list(zip(train_tokens[:-2], train_tokens[1:-1], train_tokens[2:]))

# Count trigram frequencies
trigram_counts = Counter(trigrams)

# Conditional probability P(w3 | w1, w2) = count(w1,w2,w3) / count(w1,w2)
trigram_probs = defaultdict(dict)
for (w1, w2, w3), count in trigram_counts.items():
    trigram_probs[(w1, w2)][w3] = count / bigram_counts[(w1, w2)]

# Convert to a table
trigram_table = pd.DataFrame([
    {'Prev_Words': f"{w1} {w2}", 'Next_Word': w3, 'Count': c, 'Conditional_Prob': trigram_probs[(w1, w2)][w3]}
    for (w1, w2, w3), c in trigram_counts.items()
])

# Display top 10 trigrams
print("Trigram Model Table (Top 10 trigrams):")
display(trigram_table.sort_values(by='Count', ascending=False).head(10))
```

```
Trigram Model Table (Top 10 trigrams):
```

	Prev_Words	Next_Word	Count	Conditional_Prob
126	tired sister	twice	2	0.666667
585	conversations pictures	rabbit	2	0.666667
1065	sister peeped	conversations	2	0.666667
1105	bank daisychain	pleasure	2	1.000000
442	worth considering	twice	2	0.666667
194	would reading	getting	2	0.666667
486	peeped beginning	sister	2	0.666667
983	twice suddenly	get	2	1.000000
828	peeped twice	tired	2	1.000000
25	sister sitting	close	2	0.666667

step5: apply one (laplace) smoothing

```
vocab_size = len(unigram_counts)

def laplace_bigram_prob(w1, w2):
    return (bigram_counts[(w1, w2)] + 1) / (unigram_counts[w1] + vocab_size)
```

```
"""
Smoothing is required to handle zero probabilities for unseen n-grams.
Without smoothing, any unseen word sequence would result in zero probability.
Add-one smoothing ensures every possible n-gram has a non-zero probability.
```

```
This makes the language model more robust for real-world data.
"""
```

```
'\nSmoothing is required to handle zero probabilities for unseen n-grams.\nWithout smoothing, any unseen word sequence woul
d result in zero probability.\nAdd-one smoothing ensures every possible n-gram has a non-zero probability.\nThis makes the
```

step6: sentence probability calculation

```
# Step 1: Define at least 5 example sentences
sentences = [
    "alice was tired",
    "the white rabbit ran quickly",
    "she was reading a book",
    "alice picked the daisies",
    "the rabbit had pink eyes"
]

# Print the sentences to verify
print("Selected Sentences for Probability Calculation:\n")
for s in sentences:
    print(f"- {s}")
```

```
Selected Sentences for Probability Calculation:

- alice was tired
- the white rabbit ran quickly
- she was reading a book
- alice picked the daisies
- the rabbit had pink eyes
```

```
# Step 2: Function to calculate Unigram sentence probability
def unigram_sentence_prob(sentence, unigram_probs, total_words):
    """
    Calculate the probability of a sentence using Unigram model.
    P(sentence) = product of probabilities of individual words
    """
    words = preprocess_text(sentence)  # clean and tokenize
    prob = 1.0
    for word in words:
        # Assign a small probability if the word is not in the training set
        prob *= unigram_probs.get(word, 1/total_words)
    return prob

# Test Unigram function on the first sentence
print("\nTesting Unigram Probability Function:")
print(f"Sentence: '{sentences[0]}'")
print("Probability:", unigram_sentence_prob(sentences[0], unigram_probs, total_words))
```

```

Testing Unigram Probability Function:
Sentence: 'alice was tired'
Probability: 0.0008357082816639231
```

```
# Step 3: Function to calculate bigram probability with add-one (Laplace) smoothing
def laplace_bigram_prob(w1, w2):
    return (bigram_counts[(w1, w2)] + 1) / (unigram_counts.get(w1,0) + vocab_size)

def bigram_sentence_prob(sentence):
    words = ['<s>'] + preprocess_text(sentence) + ['</s>']  # add start/end tokens
    prob = 1.0
    for i in range(len(words)-1):
        prob *= laplace_bigram_prob(words[i], words[i+1])
    return prob

# Test Bigram function on the first sentence
print("\nTesting Bigram Probability Function:")
print(f"Sentence: '{sentences[0]}'")
print("Probability:", bigram_sentence_prob(sentences[0]))
```

```

Testing Bigram Probability Function:
Sentence: 'alice was tired'
Probability: 2.0063400345090484e-05
```

```
# Step 4: Function to calculate trigram probability
def trigram_sentence_prob(sentence):
    words = ['<s>'] + preprocess_text(sentence) + ['</s>']  # add start/end tokens
    prob = 1.0
    for i in range(len(words)-2):
        w1, w2, w3 = words[i], words[i+1], words[i+2]
        # Use very small probability if trigram not seen
        prob *= trigram_probs.get((w1, w2), {}).get(w3, 1e-6)
    return prob

# Test Trigram function on the first sentence
print("\nTesting Trigram Probability Function:")
print(f"Sentence: '{sentences[0]}'")
print("Probability:", trigram_sentence_prob(sentences[0]))
```

```

Testing Trigram Probability Function:
Sentence: 'alice was tired'
Probability: 1e-12
```

```
#Step 5: Compute and display probabilities for all 5 sentences
print("\nSentence Probabilities using Unigram, Bigram, and Trigram Models:\n")
for s in sentences:
    u_prob = unigram_sentence_prob(s, unigram_probs, total_words)
    b_prob = bigram_sentence_prob(s)
    t_prob = trigram_sentence_prob(s)

    print(f"Sentence: '{s}'")
    print(f"  Unigram Probability : {u_prob:.6e}")
    print(f"  Bigram Probability  : {b_prob:.6e}")
    print(f"  Trigram Probability : {t_prob:.6e}")
    print("-"*60)
```

```

Sentence Probabilities using Unigram, Bigram, and Trigram Models:

Sentence: 'alice was tired'
  Unigram Probability : 8.357083e-04
  Bigram Probability  : 2.006340e-05
  Trigram Probability : 1.000000e-12
------------------------------------------------------------
Sentence: 'the white rabbit ran quickly'
  Unigram Probability : 1.809686e-08
  Bigram Probability  : 7.637463e-09
  Trigram Probability : 1.000000e-24
------------------------------------------------------------
Sentence: 'she was reading a book'
  Unigram Probability : 9.749930e-04
  Bigram Probability  : 4.625005e-06
  Trigram Probability : 1.000000e-12
------------------------------------------------------------
Sentence: 'alice picked the daisies'
  Unigram Probability : 7.334365e-07
  Bigram Probability  : 2.608977e-07
  Trigram Probability : 1.000000e-18
------------------------------------------------------------
Sentence: 'the rabbit had pink eyes'
  Unigram Probability : 1.998724e-05
  Bigram Probability  : 1.490472e-07
  Trigram Probability : 1.000000e-18
------------------------------------------------------------
```

step7:perplexity calculation

```
def perplexity(sentence, model_func):
    """
    Compute perplexity for a sentence using a given language model.
    Lower perplexity = model finds sentence more likely.
    """
    words = preprocess_text(sentence)  # tokenize and clean
    N = len(words)  # number of words
    prob = model_func(sentence)  # sentence probability

    # Avoid division by zero for unseen sequences
    if prob == 0:
        prob = 1e-12

    pp = pow(1/prob, 1/N)  # perplexity formula
    return pp
```

```
print("Perplexity for Sentences using Unigram, Bigram, and Trigram Models:\n")

for s in sentences:
    # Compute perplexity for each model
    u_pp = perplexity(s, lambda sent: unigram_sentence_prob(sent, unigram_probs, total_words))
    b_pp = perplexity(s, bigram_sentence_prob)
    t_pp = perplexity(s, trigram_sentence_prob)

    # Print the results neatly
    print(f"Sentence: '{s}'")
    print(f"  Unigram Perplexity : {u_pp:.2f}")
    print(f"  Bigram Perplexity  : {b_pp:.2f}")
    print(f"  Trigram Perplexity : {t_pp:.2f}")
    print("-"*60)
```

```
Perplexity for Sentences using Unigram, Bigram, and Trigram Models:

Sentence: 'alice was tired'
  Unigram Perplexity : 34.59
  Bigram Perplexity  : 223.25
  Trigram Perplexity : 1000000.00
------------------------------------------------------------
Sentence: 'the white rabbit ran quickly'
  Unigram Perplexity : 86.22
  Bigram Perplexity  : 106.97
  Trigram Perplexity : 1000000.00
------------------------------------------------------------
Sentence: 'she was reading a book'
  Unigram Perplexity : 32.03
  Bigram Perplexity  : 464.99
  Trigram Perplexity : 1000000.00
------------------------------------------------------------
Sentence: 'alice picked the daisies'
  Unigram Perplexity : 110.89
  Bigram Perplexity  : 156.50
  Trigram Perplexity : 1000000.00
------------------------------------------------------------
Sentence: 'the rabbit had pink eyes'
  Unigram Perplexity : 36.85
  Bigram Perplexity  : 188.61
  Trigram Perplexity : 1000000.00
------------------------------------------------------------
```