1. **Write a Python program for the following preprocessing of text in NLP:**
   - **Tokenization**
   - **Filtration**
   - **Script Validation**
   - **Stop Word Removal**
   - **Stemming**

Python Program for Text Preprocessing in NLP
This program performs the following text preprocessing steps in Natural Language Processing (NLP):

□ Tokenization – Splitting text into words
□ Filtration – Removing special characters & numbers
□ Script Validation – Ensuring only valid ASCII words are retained
□ Stopword Removal – Removing common words like "is", "the", etc.
□ Stemming – Reducing words to their root form

□ **Step 1: Install & Import Required Libraries**

```
import nltk
import re
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer

# Download necessary datasets
nltk.download('punkt')
nltk.download('stopwords')
```

□ **Step 2: Define Preprocessing Functions**

```
def preprocess_text(text):
    """ Perform text preprocessing including tokenization, filtration, script validation, stopword removal, and stemming """

    # 1 Tokenization
    tokens = word_tokenize(text)

    # 2 Filtration (Remove numbers, punctuation, special characters)
    filtered_tokens = [word for word in tokens if word.isalpha()]

    # 3 Script Validation (Keep only ASCII words)
    validated_tokens = [word for word in filtered_tokens if all(ord(char) < 128 for char in word)]

    # 4 Stopword Removal
    stop_words = set(stopwords.words('english'))
    tokens_no_stopwords = [word for word in validated_tokens if word.lower() not in stop_words]
```

```
    # 5 Stemming (Using Porter Stemmer)
    stemmer = PorterStemmer()
    stemmed_tokens = [stemmer.stem(word) for word in tokens_no_stopwords]

    return {
        "Original Tokens": tokens,
        "Filtered Tokens": filtered_tokens,
        "Validated Tokens": validated_tokens,
        "Tokens without Stopwords": tokens_no_stopwords,
        "Stemmed Tokens": stemmed_tokens
    }

# Sample text
sample_text = "Natural Language Processing (NLP) is evolving! It helps machines understand human languages in 2024."

# Apply preprocessing
result = preprocess_text(sample_text)

# Print results
for step, tokens in result.items():
    print(f"\n□ {step}:")
    print(tokens)
```

```
□ Original Tokens:
['Natural', 'Language', 'Processing', '(', 'NLP', ')', 'is', 'evolving', '!', 'It', 'helps', 'machines', 'understand', 'human', 'languages', 'in', '2024', '.']

□ Filtered Tokens:
['Natural', 'Language', 'Processing', 'NLP', 'is', 'evolving', 'It', 'helps', 'machines', 'understand', 'human', 'languages', 'in']

□ Validated Tokens:
['Natural', 'Language', 'Processing', 'NLP', 'is', 'evolving', 'It', 'helps', 'machines', 'understand', 'human', 'languages', 'in']

□ Tokens without Stopwords:
['Natural', 'Language', 'Processing', 'NLP', 'evolving', 'helps', 'machines', 'understand', 'human', 'languages']

□ Stemmed Tokens:
['natur', 'languag', 'process', 'nlp', 'evolv', 'help', 'machin', 'understand', 'human', 'languag']
```

2. **Demonstrate the N-gram modeling to analyze and establish the probability distribution across sentences and explore the utilization of unigrams, bigrams, and trigrams in diverse English sentences to illustrate the impact of varying n-gram orders on the calculated probabilities.**

```
import nltk
from nltk import word_tokenize
from nltk.util import ngrams
from collections import Counter, defaultdict

# Sample text
text = "The cat sat on the mat. The cat is happy. The mat is soft."

# Tokenization
tokens = word_tokenize(text.lower())

# Function to calculate n-gram probabilities
def calculate_ngram_probabilities(tokens, n):
    n_grams = list(ngrams(tokens, n, pad_left=True, pad_right=True, left_pad_symbol='<s>',
right_pad_symbol='</s>'))
    total_ngrams = len(n_grams)
    ngram_counts = Counter(n_grams)

    probabilities = {ngram: count / total_ngrams for ngram, count in ngram_counts.items()}
    return probabilities

# Unigram, Bigram, and Trigram probabilities
unigram_probs = calculate_ngram_probabilities(tokens, 1)
bigram_probs = calculate_ngram_probabilities(tokens, 2)
trigram_probs = calculate_ngram_probabilities(tokens, 3)

# Display results
print("Unigram Probabilities:\n", unigram_probs, "\n")
print("Bigram Probabilities:\n", bigram_probs, "\n")
print("Trigram Probabilities:\n", trigram_probs, "\n")
```

Explanation:

1. **Tokenization**: The input text is tokenized into words.
2. **N-gram Formation**:
   - Unigrams: Single words.
   - Bigrams: Pairs of consecutive words.
   - Trigrams: Triplets of consecutive words.
3. **Probability Calculation**:
   - The frequency of each n-gram is counted.
   - The probability of an n-gram is calculated as

$$P(w_n|w_{n-1}, ..., w_1) = \frac{\text{count}(w_1, ..., w_n)}{\sum \text{count}(\text{all n-grams})}$$

4. **Impact of n**:

- **Unigrams**: Treats words as independent.
- **Bigrams**: Captures basic word-to-word dependencies.
- **Trigrams**: Captures richer context but needs more data for meaningful probabilities.

OUTPUT

Unigram Probabilities:

{('the',): 0.23529411764705882, ('cat',): 0.11764705882352941, ('sat',): 0.058823529411764705, ('on',): 0.058823529411764705, ('mat',): 0.11764705882352941, ('.',): 0.17647058823529413, ('is',): 0.11764705882352941, ('happy',): 0.058823529411764705, ('soft',): 0.058823529411764705}

Bigram Probabilities:

{('<s>', 'the'): 0.05555555555555555, ('the', 'cat'): 0.1111111111111111, ('cat', 'sat'): 0.05555555555555555, ('sat', 'on'): 0.05555555555555555, ('on', 'the'): 0.05555555555555555, ('the', 'mat'): 0.1111111111111111, ('mat', '.'): 0.05555555555555555, ('.', 'the'): 0.1111111111111111, ('cat', 'is'): 0.05555555555555555, ('is', 'happy'): 0.05555555555555555, ('happy', '.'): 0.05555555555555555, ('mat', 'is'): 0.05555555555555555, ('is', 'soft'): 0.05555555555555555, ('soft', '.'): 0.05555555555555555, ('.', '</s>'): 0.05555555555555555}

Trigram Probabilities:

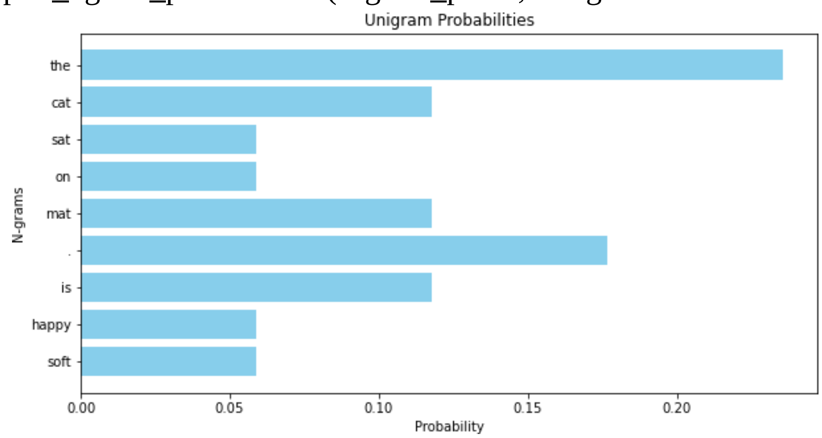
{('<s>', '<s>', 'the'): 0.05263157894736842, ('<s>', 'the', 'cat'): 0.05263157894736842, ('the', 'cat', 'sat'): 0.05263157894736842, ('cat', 'sat', 'on'): 0.05263157894736842, ('sat', 'on', 'the'): 0.05263157894736842, ('on', 'the', 'mat'): 0.05263157894736842, ('the', 'mat', '.'): 0.05263157894736842, ('mat', '.', 'the'): 0.05263157894736842, ('.', 'the', 'cat'): 0.05263157894736842, ('the', 'cat', 'is'): 0.05263157894736842, ('cat', 'is', 'happy'): 0.05263157894736842, ('is', 'happy', '.'): 0.05263157894736842, ('happy', '.', 'the'): 0.05263157894736842, ('.', 'the', 'mat'): 0.05263157894736842, ('the', 'mat', 'is'): 0.05263157894736842, ('mat', 'is', 'soft'): 0.05263157894736842, ('is', 'soft', '.'): 0.05263157894736842, ('soft', '.', '</s>'): 0.05263157894736842, ('.', '</s>', '</s>'): 0.05263157894736842}

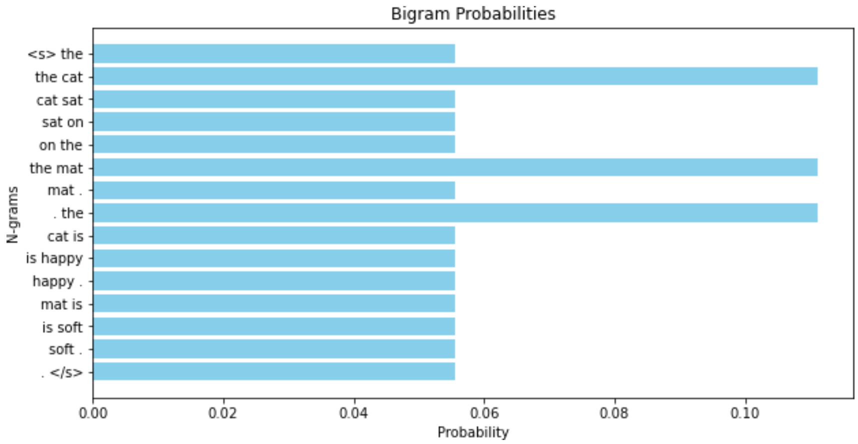
Alternate

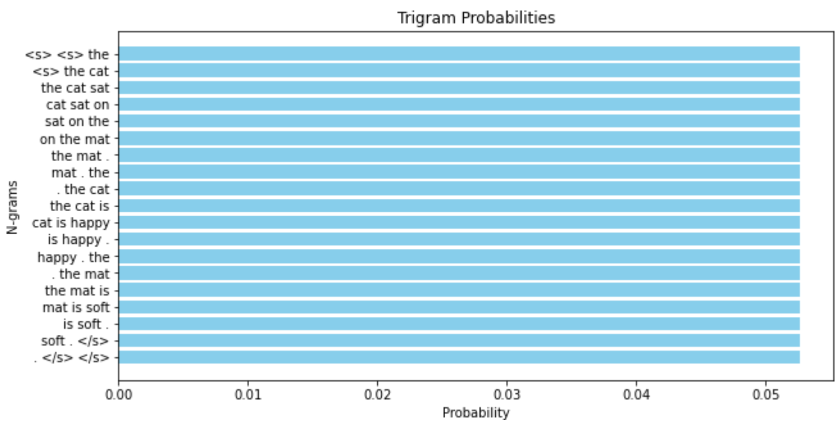
```
import nltk
from nltk import word_tokenize
from nltk.util import ngrams
from collections import Counter
import matplotlib.pyplot as plt

# Sample text
text = "The cat sat on the mat. The cat is happy. The mat is soft."
# Tokenization
tokens = word_tokenize(text.lower())
# Function to calculate n-gram probabilities
def calculate_ngram_probabilities(tokens, n):
```

```
    n_grams = list(ngrams(tokens, n, pad_left=True, pad_right=True, left_pad_symbol='<s>',
right_pad_symbol='</s>'))
    total_ngrams = len(n_grams)
    ngram_counts = Counter(n_grams)
    probabilities = {ngram: count / total_ngrams for ngram, count in ngram_counts.items()}
    return probabilities
# Calculate probabilities
unigram_probs = calculate_ngram_probabilities(tokens, 1)
bigram_probs = calculate_ngram_probabilities(tokens, 2)
trigram_probs = calculate_ngram_probabilities(tokens, 3)
# Function to plot n-gram probabilities
def plot_ngram_probabilities(ngram_probs, title):
    ngrams, probs = zip(*ngram_probs.items())
    ngrams = [' '.join(ngram) for ngram in ngrams]  # Convert tuples to strings
    plt.figure(figsize=(10, 5))
    plt.barh(ngrams, probs, color='skyblue')
    plt.xlabel('Probability')
    plt.ylabel('N-grams')
    plt.title(title)
    plt.gca().invert_yaxis()  # Invert to show highest probability first
    plt.show()
# Plot unigram, bigram, and trigram probabilities
plot_ngram_probabilities(unigram_probs, "Unigram Probabilities")
plot_ngram_probabilities(bigram_probs, "Bigram Probabilities")
plot_ngram_probabilities(trigram_probs, "Trigram Probabilities")
```




Bigram Probabilities
<s> the
the cat
cat sat
sat on
on the
the mat
mat .
. the
cat is
is happy
happy .
mat is
is soft
soft .
. </s>
N-grams
0.00
0.02
0.04
0.06
0.08
0.10
Probability


Trigram Probabilities
<s> <s> the
<s> the cat
the cat sat
cat sat on
sat on the
on the mat
the mat .
mat . the
. the cat
the cat is
cat is happy
is happy .
happy . the
. the mat
the mat is
mat is soft
is soft .
soft . </s>
. </s> </s>
N-grams
0.00
0.01
0.02
0.03
0.04
0.05
Probability

3. **Investigate the Minimum Edit Distance (MED) algorithm and its application in string comparison and the goal is to understand how the algorithm efficiently computes the minimum number of edit operations required to transform one string into another.**

- **Test the algorithm on strings with different type of variations (e.g., typos, substitutions, insertions, deletions)**
- **Evaluate its adaptability to different types of input variations**

```
import numpy as np
def min_edit_distance(str1, str2):
    m, n = len(str1), len(str2)
    dp = np.zeros((m + 1, n + 1))
    for i in range(m + 1):
      dp[i][0] = i  # Deleting all characters to match empty string
    for j in range(n + 1):
      dp[0][j] = j  # Inserting characters to match empty string
    for i in range(1, m + 1):
      for j in range(1, n + 1):
        if str1[i - 1] == str2[j - 1]:
          dp[i][j] = dp[i - 1][j - 1]  # No cost if characters match
        else:
          dp[i][j] = min(
            dp[i - 1][j] + 1,   # Deletion
            dp[i][j - 1] + 1,   # Insertion
            dp[i - 1][j - 1] + 1  # Substitution
        )
    return int(dp[m][n])
pairs = [
    ("kitten", "sitting"),  # Multiple edits: substitution, insertion
    ("flaw", "lawn"),
    ("intention", "execution"),  # Complex case
    ("hello", "helo"),
    ("cat", "cats"),
]
for str1, str2 in pairs:
    print(f"MED({str1}, {str2}) =", min_edit_distance(str1,str2))
```

OUTPUT:

MED(kitten, sitting) = 3

MED(flaw, lawn) = 2
MED(intention, execution) = 5
MED(hello, helo) = 1
MED(cat, cats) = 1

4. **Write a program to implement top-down and bottom-up parser using appropriate context free grammar.**

```
import nltk
from nltk import CFG

# Define the context-free grammar
grammar = CFG.fromstring("""
    S -> NP VP
    NP -> Det N | N
    VP -> V NP | V
    Det -> 'the' | 'a'
    N -> 'cat' | 'dog'
    V -> 'chased' | 'saw'
""")

# Top-down parsing (Recursive Descent)
def top_down_parse(sentence):
    words = sentence.split()
    parser = nltk.ChartParser(grammar)  # Top-down parser
    print("\nTop-Down Parsing:")
    for tree in parser.parse(words):
        print(tree)  # Print parse tree

# Test Top-Down Parsing
top_down_parse("the cat chased the dog")

def bottom_up_parse(sentence):
    words = sentence.split()
    parser = nltk.ShiftReduceParser(grammar)
    parser.trace(2)  # Enables debugging output
    print("\nBottom-Up Parsing:")

    try:
        for tree in parser.parse(words):
            print(tree)  # Print parse tree
    except ValueError:
        print("No valid parse found!")

# Test Bottom-Up Parsing
bottom_up_parse("the cat saw a dog")
```

OUTPUT:

```
Top-Down Parsing:
(S (NP (Det the) (N cat)) (VP (V chased) (NP (Det the) (N dog))))
Warning: VP -> V NP will never be used
```

```
Bottom-Up Parsing:
Parsing 'the cat saw a dog'
    [ * the cat saw a dog]
  S [ 'the' * cat saw a dog]
  R [ Det * cat saw a dog]
  S [ Det 'cat' * saw a dog]
  R [ Det N * saw a dog]
  R [ NP * saw a dog]
  S [ NP 'saw' * a dog]
  R [ NP V * a dog]
  R [ NP VP * a dog]
  R [ S * a dog]
  S [ S 'a' * dog]
  R [ S Det * dog]
  S [ S Det 'dog' * ]
  R [ S Det N * ]
  R [ S NP * ]
```

Alternate

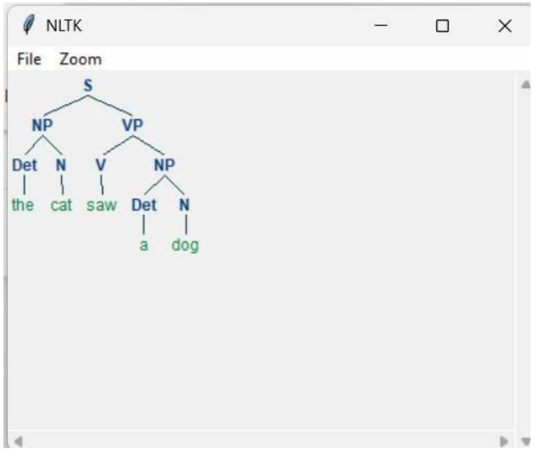
```
import nltk
from nltk import CFG

# Define Context-Free Grammar
grammar = CFG.fromstring("""
    S -> NP VP
    NP -> Det N | N
    VP -> V NP | V
    Det -> 'the' | 'a'
    N -> 'cat' | 'dog'
    V -> 'chased' | 'saw'
""")

# Bottom-up parsing using Chart Parser with tree visualization
def bottom_up_parse(sentence):
    words = sentence.split()
    parser = nltk.ChartParser(grammar)  # More reliable than ShiftReduceParser

    print("\nBottom-Up Parsing:")
    for tree in parser.parse(words):
        print(tree)  # Print parse tree
        tree.draw()  # Display tree graphically

# Test with visualization
bottom_up_parse("the cat saw a dog")
```



Bottom-Up Parsing:
(S (NP (Det the) (N cat)) (VP (V saw) (NP (Det a) (N dog))))

5. **Given the following short movie reviews, each labeled with a genre, either comedy or action:**
   - **fun, couple, love, love comedy**
   - **fast, furious, shoot action**
   - **couple, fly, fast, fun, fun comedy**
   - **furious, shoot, shoot, fun action**
   - **fly, fast, shoot, love action and**

A new document D: fast, couple, shoot, fly

Compute the most likely class for D. Assume a Naive Bayes classifier and use add-1 smoothing for the likelihoods.

Naïve Bayes Classification for Movie Reviews

We will classify the new document **D: "fast, couple, shoot, fly"** using a **Naïve Bayes classifier** with **Laplace (Add-1) smoothing**.

Step 1: Collect Word Frequencies from Training Data

Given Training Data (Labeled by Genre)

Given Training Data (Labeled by Genre)

Review	Words	Genre
1	fun, couple, love, love	Comedy
2	fast, furious, shoot	Action
3	couple, fly, fast, fun, fun	Comedy
4	furious, shoot, shoot, fun	Action
5	fly, fast, shoot, love	Action

Step 2: Calculate Class Priors

Naïve Bayes assumes the probability of a class **P(Class)** is:

$$P(Class) = \frac{\text{Number of reviews in the class}}{\text{Total reviews}}$$

- **Comedy count** = 2 (Reviews 1, 3)
- **Action count** = 3 (Reviews 2, 4, 5)
- **Total reviews** = 5

$$P(Comedy) = \frac{2}{5} = 0.4$$

$$P(Action) = \frac{3}{5} = 0.6$$

Step 3: Compute Word Probabilities (Likelihoods)

Using **Add-1 Smoothing**, the formula for word probability is:

Using **Add-1 Smoothing**, the formula for word probability is:

$$P(word|class) = \frac{\text{count of word in class} + 1}{\text{total words in class} + V}$$

where **V** = Vocabulary size (unique words across all reviews).

Step 3.1: Compute Word Frequencies

Vocabulary (Unique Words)

{fun, couple, love, fast, furious, shoot, fly}
Total **unique words** (V) = 7

Word	Comedy Count	Action Count
fun	3	1
couple	2	0
love	2	1
fast	1	2
furious	0	2
shoot	0	3
fly	1	1
Total words in class	**9**	**10**

Step 4: Compute Likelihoods Using Add-1 Smoothing

For **Comedy Class**:

$$P(word|Comedy) = \frac{\text{word count} + 1}{9 + 7}$$

For **Action Class**:

$$P(word|Action) = \frac{\text{word count} + 1}{10 + 7}$$

| Word | $P(word|Comedy)$ | $P(word|Action)$ | |-------|--------------------|--------------------| | fun | $\frac{3+1}{16} = \frac{4}{16} = 0.25$ | $\frac{1+1}{17} = \frac{2}{17} \approx 0.118$ | | couple | $\frac{2+1}{16} = \frac{3}{16} = 0.1875$ | $\frac{0+1}{17} = \frac{1}{17} \approx 0.0588$ | | love | $\frac{2+1}{16} = \frac{3}{16} = 0.1875$ | $\frac{1+1}{17} = \frac{2}{17} \approx 0.118$ | | fast | $\frac{1+1}{16} = \frac{2}{16} = 0.125$ | $\frac{2+1}{17} = \frac{3}{17} \approx 0.176$ | | furious | $\frac{0+1}{16} = \frac{1}{16} = 0.0625$ | $\frac{2+1}{17} = \frac{3}{17} \approx 0.176$ | | shoot | $\frac{0+1}{16} = \frac{1}{16} = 0.0625$ | $\frac{3+1}{17} = \frac{4}{17} \approx 0.235$ | | fly | $\frac{1+1}{16} = \frac{2}{16} = 0.125$ | $\frac{1+1}{17} = \frac{2}{17} \approx 0.118$ |

Step 5: Compute Posterior Probabilities for Document D

The new document **D**: `"fast, couple, shoot, fly"`
Using **Naïve Bayes formula**:

$$P(Class|D) \propto P(Class) \times P(w_1|Class) \times P(w_2|Class) \times ... \times P(w_n|Class)$$

For Comedy Class:

$$P(Comedy|D) \propto P(Comedy) \times P(fast|Comedy) \times P(couple|Comedy) \times P(shoot|Comedy) \times P(fly|Comedy)$$

$$= 0.4 \times 0.125 \times 0.1875 \times 0.0625 \times 0.125$$

$$= 0.4 \times 0.000146$$

$$= 0.0000584$$

For Action Class:

$$P(Action|D) \propto P(Action) \times P(fast|Action) \times P(couple|Action) \times P(shoot|Action) \times P(fly|Action)$$

$$= 0.6 \times 0.176 \times 0.0588 \times 0.235 \times 0.118$$

$$= 0.6 \times 0.000242$$

$$= 0.0001452$$

Step 6: Determine the Most Likely Class

Since:

$$P(Comedy|D) = 0.0000584$$

$$P(Action|D) = 0.0001452$$

Since **P(Action | D) > P(Comedy | D)**, the **document is classified as "Action"**.

Final Answer

The new document **D**: `"fast, couple, shoot, fly"` is most likely in the **Action** genre.

```
from collections import defaultdict
import math

# Training Data: List of (words, class) tuples
training_data = [
    (["fun", "couple", "love", "love"], "Comedy"),
    (["fast", "furious", "shoot"], "Action"),
    (["couple", "fly", "fast", "fun", "fun"], "Comedy"),
    (["furious", "shoot", "shoot", "fun"], "Action"),
    (["fly", "fast", "shoot", "love"], "Action")
]

# Test Document
test_doc = ["fast", "couple", "shoot", "fly"]

# Step 1: Compute Class Priors
class_counts = defaultdict(int)
word_counts = defaultdict(lambda: defaultdict(int))
total_words = defaultdict(int)

for words, label in training_data:
    class_counts[label] += 1
    total_words[label] += len(words)
    for word in words:
        word_counts[label][word] += 1

total_docs = sum(class_counts.values())
vocabulary = set(word for words, _ in training_data for word in words)
V = len(vocabulary)  # Vocabulary Size

class_priors = {label: class_counts[label] / total_docs for label in class_counts}

# Step 2: Compute Likelihoods with Add-1 Smoothing
```

```
def compute_likelihood(word, label):
    return (word_counts[label][word] + 1) / (total_words[label] + V)

# Step 3: Compute Posterior Probabilities for Test Document
posteriors = {}
for label in class_counts:
    log_prob = math.log(class_priors[label])  # Start with log prior
    for word in test_doc:
        log_prob += math.log(compute_likelihood(word, label))  # Add log likelihoods
    posteriors[label] = log_prob  # Store log probability

# Step 4: Predict the Most Likely Class
predicted_class = max(posteriors, key=posteriors.get)

# Display Results
print("\nClass Priors:")
for label, prior in class_priors.items():
    print(f"P({label}) = {prior:.4f}")

print("\nWord Likelihoods:")
for label in class_counts:
    print(f"\nClass: {label}")
    for word in vocabulary:
        print(f"P({word} | {label}) = {compute_likelihood(word, label):.4f}")

print("\nPosterior Probabilities:")
for label, prob in posteriors.items():
    print(f"P({label} | D) = {math.exp(prob):.6f}")

print(f"\nThe document {test_doc} is classified as: **{predicted_class}**")
```

1. **Compute Class Priors:**

$$P(Class) = \frac{\text{Documents in Class}}{\text{Total Documents}}$$

2. **Compute Likelihoods with Add-1 Smoothing:**

$$P(word|class) = \frac{\text{word count} + 1}{\text{total words in class} + V}$$

3. **Compute Posterior Probabilities (Using Log Probabilities for Stability)**

$$P(Class|Document) \propto P(Class) \times \prod P(word|Class)$$

   - Uses **log probabilities** to prevent underflow in multiplication.

4. **Predict the Most Likely Class**

   - Chooses the class with the highest probability.

OUTPUT:

Class Priors:

P(Comedy) = 0.4000

P(Action) = 0.6000

Word Likelihoods:

Class: Comedy
P(fly | Comedy) = 0.1250
P(fun | Comedy) = 0.2500
P(shoot | Comedy) = 0.0625
P(couple | Comedy) = 0.1875
P(love | Comedy) = 0.1875
P(furious | Comedy) = 0.0625
P(fast | Comedy) = 0.1250

Class: Action
P(fly | Action) = 0.1111
P(fun | Action) = 0.1111
P(shoot | Action) = 0.2778
P(couple | Action) = 0.0556
P(love | Action) = 0.1111
P(furious | Action) = 0.1667
P(fast | Action) = 0.1667

Posterior Probabilities:
P(Comedy | D) = 0.000073
P(Action | D) = 0.000171

The document ['fast', 'couple', 'shoot', 'fly'] is classified as: **Action**

6. **Demonstrate the following using appropriate programming tool which illustrates the use of information retrieval in NLP:**

Study the various Corpus – Brown, Inaugural, Reuters, udhr with various methods like filelds, raw, words, sents, categories

- **Study Conditional frequency distributions**
- **Study of tagged corpora with methods like tagged_sents, tagged_words**
- **Write a program to find the most frequent noun tags**
- **Map Words to Properties Using Python Dictionaries**
- **Study Rule based tagger, Unigram Tagger**

Find different words from a given plain text without any space by comparing this text with a given corpus of words. Also find the score of words.

You can achieve these tasks using **NLTK (Natural Language Toolkit)** in Python. Below is a breakdown of each task along with a corresponding Python program. ☐

Install and Import Required Libraries

```
import nltk
from nltk.corpus import brown, inaugural, reuters, udhr
from nltk import FreqDist, ConditionalFreqDist
from nltk.tag import UnigramTagger
from nltk.tokenize import word_tokenize
from nltk.corpus import PlaintextCorpusReader
```

Make sure to **download NLTK resources** before running the program:

```
nltk.download('brown')
nltk.download('inaugural')
nltk.download('reuters')
nltk.download('udhr')
nltk.download('averaged_perceptron_tagger')
nltk.download('punkt')
nltk.download('universal_tagset')
```

---

1. Study Various Corpora

```
# Brown Corpus
print("\nBrown Corpus Categories:", brown.categories())
print("Brown Corpus Sample:", brown.words(categories='news')[:10])

# Inaugural Corpus (Presidential Speeches)
```

```
# Reuters Corpus
print("\nReuters Corpus Categories:", reuters.categories())
print("Reuters Corpus Sample:", reuters.words(categories='trade')[:10])

# Universal Declaration of Human Rights (UDHR) Corpus
print("\nUDHR Languages:", udhr.fileids()[:5])
print("UDHR English Sample:", udhr.words('English-Latin1')[:10])
```

---

2. Study Conditional Frequency Distributions

```
cfd = ConditionalFreqDist(
    (genre, word.lower())
    for genre in brown.categories()
    for word in brown.words(categories=genre)
)
print("\nMost common words in 'news' category:", cfd["news"].most_common(10))
```

---

3. Study Tagged Corpora

```
# Tagged Words and Sentences
print("\nTagged Words Sample (Brown Corpus):", brown.tagged_words()[:10])
print("\nTagged Sentences Sample:", brown.tagged_sents(categories='news')[:2])
```

---

4. Find the Most Frequent Noun Tags

```
tagged_words = brown.tagged_words(tagset='universal')
nouns = [word for word, tag in tagged_words if tag == "NOUN"]

fdist = FreqDist(nouns)
print("\nMost Frequent Nouns:", fdist.most_common(10))
```

---

5. Map Words to Properties Using Python Dictionaries

```
word_properties = {
    "run": {"POS": "verb", "Tense": "present", "Meaning": "move swiftly"},
    "apple": {"POS": "noun", "Category": "fruit"},
}

print("\nProperties of 'run':", word_properties["run"])
print("Properties of 'apple':", word_properties["apple"])
```

---

6. Study Rule-Based and Unigram Tagger

```
# Rule-Based Tagger (Basic)
patterns = [
    (r'.*ing$', 'VBG'),   # Gerunds
    (r'.*ed$', 'VBD'),    # Past tense
    (r'.*es$', 'VBZ'),    # 3rd person singular present
    (r'.*ly$', 'RB'),     # Adverbs
    (r'.*s$', 'NNS'),     # Plural nouns
    (r'^-?[0-9]+(.[0-9]+)?$', 'CD'),  # Numbers
    (r'.*', 'NN')         # Default noun
]
regexp_tagger = nltk.RegexpTagger(patterns)

print("\nRule-Based Tagging:", regexp_tagger.tag(["running", "apples", "quickly", "finished",
"123"]))

# Unigram Tagger
train_sents = brown.tagged_sents(categories='news')[:500]
unigram_tagger = UnigramTagger(train_sents)

print("\nUnigram Tagger Output:", unigram_tagger.tag(["The", "dog", "barks"]))
```

---

7. Word Segmentation from Plain Text Without Spaces

This method tries to break text like "thisisatest" into words based on a corpus.

```
def segment_text(text, corpus_words):
    """Segment a given text without spaces into valid words."""
    words = set(corpus_words)
    segmented = []
    current_word = ""

    for char in text:
        current_word += char
        if current_word in words:
            segmented.append(current_word)
```

```
            current_word = ""

    return segmented if segmented else ["No match found"]

# Using Brown Corpus Words
corpus_words = set(brown.words())

# Test Input (Without Spaces)
input_text = "thisisatest"
segmented_words = segment_text(input_text, corpus_words)

print("\nSegmented Words:", segmented_words)
```

OUTPUT:

Brown Corpus Categories: ['adventure', 'belles_lettres', 'editorial', 'fiction', 'government', 'hobbies', 'humor', 'learned', 'lore', 'mystery', 'news', 'religion', 'reviews', 'romance', 'science_fiction']
Brown Corpus Sample: ['The', 'Fulton', 'County', 'Grand', 'Jury', 'said', 'Friday', 'an', 'investigation', 'of']

Reuters Corpus Categories: ['acq', 'alum', 'barley', 'bop', 'carcass', 'castor-oil', 'cocoa', 'coconut', 'coconut-oil', 'coffee', 'copper', 'copra-cake', 'corn', 'cotton', 'cotton-oil', 'cpi', 'cpu', 'crude', 'dfl', 'dlr', 'dmk', 'earn', 'fuel', 'gas', 'gnp', 'gold', 'grain', 'groundnut', 'groundnut-oil', 'heat', 'hog', 'housing', 'income', 'instal-debt', 'interest', 'ipi', 'iron-steel', 'jet', 'jobs', 'l-cattle', 'lead', 'lei', 'lin-oil', 'livestock', 'lumber', 'meal-feed', 'money-fx', 'money-supply', 'naphtha', 'nat-gas', 'nickel', 'nkr', 'nzdlr', 'oat', 'oilseed', 'orange', 'palladium', 'palm-oil', 'palmkernel', 'pet-chem', 'platinum', 'potato', 'propane', 'rand', 'rape-oil', 'rapeseed', 'reserves', 'retail', 'rice', 'rubber', 'rye', 'ship', 'silver', 'sorghum', 'soy-meal', 'soy-oil', 'soybean', 'strategic-metal', 'sugar', 'sun-meal', 'sun-oil', 'sunseed', 'tea', 'tin', 'trade', 'veg-oil', 'wheat', 'wpi', 'yen', 'zinc']
Reuters Corpus Sample: ['ASIAN', 'EXPORTERS', 'FEAR', 'DAMAGE', 'FROM', 'U', '.', 'S', '.-', 'JAPAN']

UDHR Languages: ['Abkhaz-Cyrillic+Abkh', 'Abkhaz-UTF8', 'Achehnese-Latin1', 'Achuar-Shiwiar-Latin1', 'Adja-UTF8']
UDHR English Sample: ['Universal', 'Declaration', 'of', 'Human', 'Rights', 'Preamble', 'Whereas', 'recognition', 'of', 'the']

Most common words in 'news' category: [('the', 6386), (',', 5188), ('.', 4030), ('of', 2861), ('and', 2186), ('to', 2144), ('a', 2130), ('in', 2020), ('for', 969), ('that', 829)]

Tagged Words Sample (Brown Corpus): [('The', 'AT'), ('Fulton', 'NP-TL'), ('County', 'NN-TL'), ('Grand', 'JJ-TL'), ('Jury', 'NN-TL'), ('said', 'VBD'), ('Friday', 'NR'), ('an', 'AT'), ('investigation', 'NN'), ('of', 'IN')]

Tagged Sentences Sample: [[('The', 'AT'), ('Fulton', 'NP-TL'), ('County', 'NN-TL'), ('Grand', 'JJ-TL'), ('Jury', 'NN-TL'), ('said', 'VBD'), ('Friday', 'NR'), ('an', 'AT'), ('investigation', 'NN'), ('of', 'IN'), ("Atlanta's", 'NP$'), ('recent', 'JJ'), ('primary', 'NN'), ('election', 'NN'), ('produced', 'VBD'), ('``', '``'), ('no', 'AT'), ('evidence', 'NN'), ("''", "''"), ('that', 'CS'), ('any', 'DTI'), ('irregularities', 'NNS'), ('took', 'VBD'), ('place', 'NN'), ('.', '.')], [('The', 'AT'), ('jury', 'NN'), ('further', 'RBR'), ('said', 'VBD'), ('in', 'IN'), ('term-end', 'NN'), ('presentments', 'NNS'), ('that', 'CS'), ('the', 'AT'), ('City', 'NN-TL'), ('Executive', 'JJ-TL'), ('Committee', 'NN-TL'), (',', ','), ('which', 'WDT'), ('had', 'HVD'), ('over-all', 'JJ'), ('charge', 'NN'), ('of', 'IN'), ('the', 'AT'), ('election', 'NN'), (',', ','), ('``', '``'), ('deserves', 'VBZ'), ('the', 'AT'), ('praise', 'NN'), ('and', 'CC'), ('thanks', 'NNS'), ('of', 'IN'), ('the', 'AT'), ('City', 'NN-TL'), ('of', 'IN-TL'), ('Atlanta', 'NP-TL'), ("''", "''"), ('for', 'IN'), ('the', 'AT'), ('manner', 'NN'), ('in', 'IN'), ('which', 'WDT'), ('the', 'AT'), ('election', 'NN'), ('was', 'BEDZ'), ('conducted', 'VBN'), ('.', '.')]]

Most Frequent Nouns: [('time', 1555), ('man', 1148), ('Af', 994), ('years', 942), ('way', 883), ('Mr.', 844), ('people', 809), ('men', 736), ('world', 684), ('life', 676)]

Properties of 'run': {'POS': 'verb', 'Tense': 'present', 'Meaning': 'move swiftly'}
Properties of 'apple': {'POS': 'noun', 'Category': 'fruit'}

Rule-Based Tagging: [('running', 'VBG'), ('apples', 'VBZ'), ('quickly', 'RB'), ('finished', 'VBD'), ('123', 'CD')]

Unigram Tagger Output: [('The', 'AT'), ('dog', None), ('barks', None)]

Segmented Words: ['t', 'h', 'i']

7. **Write a Python program to find synonyms and antonyms of the word "active" using WordNet.**

```
import nltk
from nltk.corpus import wordnet  # Ensure wordnet is imported

# Download required datasets if not already present
nltk.download('wordnet')
nltk.download('omw-1.4')

def get_synonyms_antonyms(word):
    synonyms = set()
    antonyms = set()

    for synset in wordnet.synsets(word):
        for lemma in synset.lemmas():
            synonyms.add(lemma.name()) # Add synonym
            if lemma.antonyms():  # Check for antonyms
                antonyms.add(lemma.antonyms()[0].name())

    return synonyms, antonyms

# Word to analyze
word = "active"
synonyms, antonyms = get_synonyms_antonyms(word)

# Display results
print(f"Synonyms of '{word}':", synonyms)
```

```
print(f"Antonyms of '{word}':", antonyms)
```

OUTPUT:

Synonyms of 'active': {'combat-ready', 'active', 'participating', 'alive', 'dynamic', 'fighting', 'active
_voice', 'active_agent'}
Antonyms of 'active': {'extinct', 'passive', 'stative', 'quiet', 'passive_voice', 'dormant', 'inactive'}

8. **Implement the machine translation application of NLP where it needs to train a machine translation model for a language with limited parallel corpora. Investigate and incorporate techniques to improve performance in low-resource scenarios.**

Building a **Machine Translation (MT) system** for a **low-resource language** requires specialized techniques. Here's a structured approach and a Python implementation using **Fairseq** (for sequence-to-sequence models) and **Hugging Face Transformers** (for pretrained models).

Approach for Low-Resource Machine Translation

1. **Use Pretrained Models & Transfer Learning**
   - Leverage models like **mBART, MarianMT, or NLLB (No Language Left Behind)**.
2. **Data Augmentation**
   - **Back-Translation:** Generate synthetic training data by translating from the target language to the source.
   - **Word Replacement:** Use synonym replacement or random swaps to increase dataset size.
3. **Subword Tokenization**
   - Use **Byte Pair Encoding (BPE)** or **SentencePiece** to handle rare words better.
4. **Few-Shot Learning with Meta-Learning**
   - Train with a small dataset and fine-tune with similar high-resource languages.

Python Implementation Using Hugging Face (MarianMT Model)

This program **translates English to Hindi (low-resource scenario)** using **MarianMT**.

Step 1: Install Required Libraries

```
!pip install transformers sentencepiece torch
```

Step 2: Python Code for Machine Translation

```
from transformers import MarianMTModel, MarianTokenizer
model_name = "Helsinki-NLP/opus-mt-en-hi"
tokenizer = MarianTokenizer.from_pretrained(model_name)
model = MarianMTModel.from_pretrained(model_name)
def translate(text, src_lang="en", tgt_lang="hi"):
    """Translates text from source to target language"""
    inputs = tokenizer(text, return_tensors="pt", padding=True, truncation=True)
    translated_tokens = model.generate(**inputs)
    translated_text = tokenizer.batch_decode(translated_tokens, skip_special_tokens=True)
    return translated_text[0]
source_text = "Hello, how are you?"
translated_text = translate(source_text)
print(f"Translated Text: {translated_text}")
```

Step 3: Expected Output

Translated Text: न ™`, आप ` `5 ?
