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

Program:

```
import nltk
import re
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer
# Download necessary NLTK resources
nltk.download('punkt')
nltk.download('stopwords')

def preprocess_text(text):
    text = text.replace('\u00A0', ' ')
   # Step 1: Tokenization
   tokens = word_tokenize(text)
   print("Tokens:", tokens)

   # Step 2: Filtration (remove special characters, numbers, etc.)
   filtered_tokens = [word for word in tokens if re.match(r'^[a-zA-Z]+$', word)]
   print("Filtered Tokens:", filtered_tokens)

   # Step 3: Script Validation (ensure all tokens are in English script)
   # Assuming the text is already in English, no further action is needed.
   # If not, you can use a language detection library like `langdetect`.

   # Step 4: Stop Word Removal
   stop_words = set(stopwords.words('english'))
   tokens_without_stopwords = [word for word in filtered_tokens if word.lower() not
in stop_words]
   print("Tokens without Stopwords:", tokens_without_stopwords)

   # Step 5: Stemming
   stemmer = PorterStemmer()
   stemmed_tokens = [stemmer.stem(word) for word in tokens_without_stopwords]
   print("Stemmed Tokens:", stemmed_tokens)

   return stemmed_tokens

# Example Usage
text = "This is an example text! It includes different words, numbers like 123, and
punctuation."
processed_text = preprocess_text(text)
print("Processed Tokens:", processed_text)
```

Output:

Tokens: ['This', 'is', 'an', 'example', 'text', '!', 'It', 'includes', 'different', 'words', ',', 'numbers', 'like', '123', ',', 'and', 'punctuation', '.']
Filtered Tokens: ['This', 'is', 'an', 'example', 'text', 'It', 'includes', 'different', 'words', 'numbers', 'like', 'and', 'punctuation']
Tokens without Stopwords: ['example', 'text', 'includes', 'different', 'words', 'numbers', 'like', 'punctuation']
Stemmed Tokens: ['exampl', 'text', 'includ', 'differ', 'word', 'number', 'like', 'punctuat']
Processed Tokens: ['exampl', 'text', 'includ', 'differ', 'word', 'number', 'like', 'punctuat']

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.

- Unigrams (n=1): Single words (e.g., "quick", "brown", "fox").
- Bigrams (n=2): Pairs of consecutive words (e.g., "quick brown", "brown fox").
- Trigrams (n=3): Triplets of consecutive words (e.g., "quick brown fox").

Steps:

1. Tokenize sentences into unigrams, bigrams, and trigrams.
2. Calculate the probability distribution of these N-grams.
3. Analyze how the order of N-grams affects the probabilities.

Program:

```
import nltk
from nltk.util import ngrams
from collections import Counter
from nltk.tokenize import word_tokenize
from nltk.probability import FreqDist
# Download necessary NLTK resources
nltk.download('punkt_tab')

# Sample sentences
sentences = [
    "The quick brown fox jumps over the lazy dog.",
    "A quick brown fox jumps over the lazy dog.",
    "The lazy dog is jumped over by the quick brown fox."
]

# Function to generate N-grams and calculate probabilities
def ngram_probability(sentences, n):
    # Tokenize sentences and generate N-grams
    tokens = []
    for sentence in sentences:
        tokens.extend(word_tokenize(sentence.lower()))

    # Generate N-grams
    n_grams = list(ngrams(tokens, n))

    # Calculate frequency distribution
    freq_dist = FreqDist(n_grams)

    # Calculate probabilities
    total_ngrams = len(n_grams)
    probabilities = {gram: count / total_ngrams for gram, count in freq_dist.items()}

    return probabilities
```

```
# Unigrams (n=1)
unigram_probs = ngram_probability(sentences, 1)
print("Unigram Probabilities:")
for gram, prob in unigram_probs.items():
    print(f"{gram}: {prob:.4f}")

# Bigrams (n=2)
bigram_probs = ngram_probability(sentences, 2)
print("\nBigram Probabilities:")
for gram, prob in bigram_probs.items():
    print(f"{gram}: {prob:.4f}")


# Trigrams (n=3)
trigram_probs = ngram_probability(sentences, 3)
print("\nTrigram Probabilities:")
for gram, prob in trigram_probs.items():
    print(f"{gram}: {prob:.4f}")
```

Output:

```
Unigram Probabilities:
('the',): 0.1562
('quick',): 0.0938
('brown',): 0.0938
('fox',): 0.0938
('jumps',): 0.0625
('over',): 0.0938
('lazy',): 0.0938
('dog',): 0.0938
('.',): 0.0938
('a',): 0.0312
('is',): 0.0312
('jumped',): 0.0312
('by',): 0.0312


Bigram Probabilities:
('the', 'quick'): 0.0645
('quick', 'brown'): 0.0968
('brown', 'fox'): 0.0968
('fox', 'jumps'): 0.0645
('jumps', 'over'): 0.0645
('over', 'the'): 0.0645
('the', 'lazy'): 0.0968
('lazy', 'dog'): 0.0968
('dog', '.'): 0.0645
('.', 'a'): 0.0323
('a', 'quick'): 0.0323
```

('.', 'the'): 0.0323
('dog', 'is'): 0.0323
('is', 'jumped'): 0.0323
('jumped', 'over'): 0.0323
('over', 'by'): 0.0323
('by', 'the'): 0.0323
('fox', '.'): 0.0323

Trigram Probabilities:
('the', 'quick', 'brown'): 0.0667
('quick', 'brown', 'fox'): 0.1000
('brown', 'fox', 'jumps'): 0.0667
('fox', 'jumps', 'over'): 0.0667
('jumps', 'over', 'the'): 0.0667
('over', 'the', 'lazy'): 0.0667
('the', 'lazy', 'dog'): 0.1000
('lazy', 'dog', '.'): 0.0667
('dog', '.', 'a'): 0.0333
('.', 'a', 'quick'): 0.0333
('a', 'quick', 'brown'): 0.0333
('dog', '.', 'the'): 0.0333
('.', 'the', 'lazy'): 0.0333
('lazy', 'dog', 'is'): 0.0333
('dog', 'is', 'jumped'): 0.0333
('is', 'jumped', 'over'): 0.0333
('jumped', 'over', 'by'): 0.0333
('over', 'by', 'the'): 0.0333
('by', 'the', 'quick'): 0.0333
('brown', 'fox', '.'): 0.0333

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

Program:

```
def min_edit_distance(str1, str2):
    m = len(str1)
    n = len(str2)

    # Create a DP table to store results of subproblems
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    # Initialize the base cases
    for i in range(m + 1):
        dp[i][0] = i  # Deletion cost
    for j in range(n + 1):
        dp[0][j] = j  # Insertion cost

    # Fill the DP table
    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 operation needed
            else:
                dp[i][j] = 1 + min(
                    dp[i - 1][j],  # Deletion
                    dp[i][j - 1],  # Insertion
                    dp[i - 1][j - 1]  # Substitution
                )

    # The final result is in dp[m][n]
    return dp[m][n]

# Test cases
test_cases = [
    ("kitten", "sitting"),  # Substitutions and insertions
    ("intention", "execution"),  # Substitutions and deletions
    ("flaw", "lawn"),  # Substitutions
    ("apple", "aple"),  # Deletion
    ("book", "books"),  # Insertion
    ("abc", "def"),  # All substitutions
    ("", "abc"),  # All insertions
    ("abc", "")  # All deletions
]

# Evaluate MED for each test case
for str1, str2 in test_cases:
```

```
distance = min_edit_distance(str1, str2)
print(f"MED between '{str1}' and '{str2}': {distance}")
```

Output:

MED between 'kitten' and 'sitting': 3
MED between 'intention' and 'execution': 5
MED between 'flaw' and 'lawn': 2
MED between 'apple' and 'aple': 1
MED between 'book' and 'books': 1
MED between 'abc' and 'def': 3
MED between '' and 'abc': 3
MED between 'abc' and '': 3

Explanation:

The **Minimum Edit Distance (MED)** algorithm is a dynamic programming approach used to measure the similarity between two strings. It calculates the minimum number of operations required to transform one string into another.

1. **Substitutions and Insertions**:
   - "kitten" → "sitting": Replace 'k' with 's', replace 'e' with 'i', and insert 'g'.
   - MED = 3.
2. **Substitutions and Deletions**:
   - "intention" → "execution": Replace 'i' with 'e', replace 'n' with 'x', delete 'n'.
   - MED = 5.
3. **Substitutions**:
   - "flaw" → "lawn": Replace 'f' with 'l', replace 'w' with 'n'.
   - MED = 2.
4. **Deletion**:
   - "apple" → "aple": Delete 'p'.
   - MED = 1.
5. **Insertion**:
   - "book" → "books": Insert 's'.
   - MED = 1.
6. **All Substitutions**:
   - "abc" → "def": Replace all characters.
   - MED = 3.
7. **All Insertions**:
   - "" → "abc": Insert all characters.
   - MED = 3.
8. **All Deletions**:
   - "abc" → "": Delete all characters.
   - MED = 3.

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

Program:

```
import nltk
```

```
from nltk import CFG

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

# Create Top-Down (Recursive Descent) and Bottom-Up (Chart) parsers
top_down_parser = nltk.RecursiveDescentParser(grammar)
bottom_up_parser = nltk.ChartParser(grammar)

# Input sentence
sentence = "the cat chased a dog".split()

# Top-Down Parsing
print("Top-Down Parsing Results:")
for tree in top_down_parser.parse(sentence):
    print(tree)

# Bottom-Up Parsing
print("\nBottom-Up Parsing Results:")
for tree in bottom_up_parser.parse(sentence):
    print(tree)
```

Output:

```
Top-Down Parsing Results:
(S (NP (Det the) (N cat)) (VP (V chased) (NP (Det a) (N dog))))

Bottom-Up Parsing Results:
(S (NP (Det the) (N cat)) (VP (V chased) (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.

Program:

```
from collections import defaultdict
import math

def train_naive_bayes(data):
    class_counts = defaultdict(int)
    word_counts = defaultdict(lambda: defaultdict(int))
    vocab = set()

    # Count occurrences
    for words, label in data:
        class_counts[label] += 1
        for word in words:
            word_counts[label][word] += 1
            vocab.add(word)

    return class_counts, word_counts, vocab

def calculate_probabilities(class_counts, word_counts, vocab, text, alpha=1):
    total_reviews = sum(class_counts.values())
    probabilities = {}

    for label in class_counts:
        # Prior probability: P(Class)
        prob = math.log(class_counts[label] / total_reviews)
        total_words = sum(word_counts[label].values())
        vocab_size = len(vocab)

        # Compute likelihood with add-1 smoothing: P(w|Class)
        for word in text:
            word_freq = word_counts[label][word] + alpha
            prob += math.log(word_freq / (total_words + vocab_size * alpha))

        probabilities[label] = prob

    return probabilities

def classify(class_counts, word_counts, vocab, text):
    probabilities = calculate_probabilities(class_counts, word_counts, vocab, text)
```

```
    return max(probabilities, key=probabilities.get)

# Training Data
reviews = [
    (['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')
]

# Train Naive Bayes Classifier
class_counts, word_counts, vocab = train_naive_bayes(reviews)

# New document
D = ['fast', 'couple', 'shoot', 'fly']

# Classify new document
predicted_class = classify(class_counts, word_counts, vocab, D)
print(predicted_class)
```

Output:

Action