**1.Write a Python program for the following preprocessing of text in NLP**

**: ● Tokenization**

**● Filtration**

**● Script Validation**

**● Stop Word Removal**

**● Stemming**

**import nltk**

**import re**

**import string**

**from nltk.tokenize import word\_tokenize**

**from nltk.corpus import stopwords**

**from nltk.stem import PorterStemmer**

**# Download required NLTK datasets**

**nltk.download('punkt')**

**nltk.download('stopwords')**

**def preprocess\_text(text):**

**# Tokenization: Split the text into words (tokens)**

**tokens = word\_tokenize(text)**

**print(f"Tokens: {tokens}")**

**# Filtration: Remove non-alphabetic tokens and punctuation**

**tokens = [word for word in tokens if word.isalpha()]**

**print(f"Filtered Tokens (Alphabetic only): {tokens}")**

**# Script Validation: Check if the text is in a valid script (English letters)**

**# For now, we'll just allow English alphabet and basic punctuation**

**if not all(char.isalpha() or char.isspace() for char in text):**

**raise ValueError("Text contains invalid characters outside the English alphabet or basic punctuation.")**

**# Stop Word Removal: Remove common words that don't add much meaning**

**stop\_words = set(stopwords.words('english'))**

**filtered\_tokens = [word for word in tokens if word.lower() not in stop\_words]**

**print(f"Tokens after Stop Word Removal: {filtered\_tokens}")**

**# Stemming: Reduce words to their root form (e.g., "running" → "run")**

**stemmer = PorterStemmer()**

**stemmed\_tokens = [stemmer.stem(word) for word in filtered\_tokens]**

**print(f"Stemmed Tokens: {stemmed\_tokens}")**

**return stemmed\_tokens**

**# Example input text**

**text = "Running is good for health. The quick brown fox jumps over the lazy dog."**

**# Preprocess the text**

**preprocessed\_tokens = preprocess\_text(text)**

**print("\nFinal Processed Tokens:", preprocessed\_tokens)**

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.util import ngrams**

**from collections import Counter**

**# Sample sentence**

**sentence = "I enjoy learning natural language processing"**

**# Tokenize the sentence into words**

**tokens = nltk.word\_tokenize(sentence.lower()) # Lowercase for uniformity**

**print(f"Tokens: {tokens}")**

**# Function to calculate n-gram probabilities**

**def calculate\_ngram\_probabilities(tokens, n):**

**# Create the n-grams**

**n\_grams = list(ngrams(tokens, n))**

**# Count frequencies of n-grams**

**n\_gram\_freq = Counter(n\_grams)**

**total\_ngrams = sum(n\_gram\_freq.values())**

**# Calculate probabilities for each n-gram**

**n\_gram\_probabilities = {ngram: freq / total\_ngrams for ngram, freq in n\_gram\_freq.items()}**

**return n\_gram\_probabilities**

**# Calculate unigram (1-gram) probabilities**

**unigram\_probs = calculate\_ngram\_probabilities(tokens, 1)**

**print("\nUnigram Probabilities:")**

**for unigram, prob in unigram\_probs.items():**

**print(f"{' '.join(unigram)}: {prob:.4f}")**

**# Calculate bigram (2-gram) probabilities**

**bigram\_probs = calculate\_ngram\_probabilities(tokens, 2)**

**print("\nBigram Probabilities:")**

**for bigram, prob in bigram\_probs.items():**

**print(f"{' '.join(bigram)}: {prob:.4f}")**

**# Calculate trigram (3-gram) probabilities**

**trigram\_probs = calculate\_ngram\_probabilities(tokens, 3)**

**print("\nTrigram Probabilities:")**

**for trigram, prob in trigram\_probs.items():**

**print(f"{' '.join(trigram)}: {prob:.4f}")**

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

**def min\_edit\_distance(source, target):**

**# Initialize a DP table with dimensions (len(source) + 1) x (len(target) + 1)**

**m, n = len(source), len(target)**

**dp = [[0] \* (n + 1) for \_ in range(m + 1)]**

**# Initialize the base cases**

**for i in range(m + 1):**

**dp[i][0] = i # Deleting all characters from source to match empty target**

**for j in range(n + 1):**

**dp[0][j] = j # Inserting all characters of target into an empty source**

**# Fill the DP table**

**for i in range(1, m + 1):**

**for j in range(1, n + 1):**

**# If the characters match, no operation is required**

**if source[i - 1] == target[j - 1]:**

**dp[i][j] = dp[i - 1][j - 1]**

**else:**

**# If characters don't match, consider insertion, deletion, and substitution**

**dp[i][j] = min(dp[i - 1][j] + 1, # Deletion**

**dp[i][j - 1] + 1, # Insertion**

**dp[i - 1][j - 1] + 1) # Substitution**

**return dp[m][n]**

**# Test the algorithm with different types of variations**

**test\_cases = [**

**("kitten", "sitting"), # Substitution and insertion**

**("flaw", "lawn"), # Substitution**

**("intention", "execution"), # Substitution, insertion, and deletion**

**("abcdef", "abdf"), # Deletion and insertion**

**("hello", "helo") # Deletion**

**]**

**for source, target in test\_cases:**

**print(f"Edit Distance between '{source}' and '{target}': {min\_edit\_distance(source, target)}")**

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

**class TopDownParser:**

**def \_\_init\_\_(self, input\_string):**

**self.input\_string = input\_string**

**self.pos = 0**

**def parse(self):**

**if self.S():**

**if self.pos == len(self.input\_string):**

**print("String accepted by the grammar.")**

**else:**

**print("Parsing failed. Unused input detected.")**

**else:**

**print("Parsing failed.")**

**def S(self):**

**# S → A B**

**print("Attempting to match S → A B")**

**if self.A():**

**if self.B():**

**return True**

**return False**

**def A(self):**

**# A → a**

**print("Attempting to match A → a")**

**if self.pos < len(self.input\_string) and self.input\_string[self.pos] == 'a':**

**self.pos += 1**

**return True**

**return False**

**def B(self):**

**# B → b**

**print("Attempting to match B → b")**

**if self.pos < len(self.input\_string) and self.input\_string[self.pos] == 'b':**

**self.pos += 1**

**return True**

**return False**

**# Test the Top-Down Parser**

**input\_string = "ab"**

**parser = TopDownParser(input\_string)**

**parser.parse()**

b. class BottomUpParser:

def \_\_init\_\_(self, input\_string):

self.input\_string = input\_string

self.stack = []

self.pos = 0

def parse(self):

while self.pos < len(self.input\_string) or len(self.stack) > 0:

print(f"Stack: {self.stack}, Remaining Input: {self.input\_string[self.pos:]}")

# Shift operation

if self.pos < len(self.input\_string):

self.stack.append(self.input\_string[self.pos])

self.pos += 1

# Try to reduce

if self.reduce():

continue

# If no reduction is possible and input is exhausted, break

if len(self.stack) > 0 and self.pos == len(self.input\_string):

print("Parsing failed.")

return

print("String accepted by the grammar.")

def reduce(self):

# Check if we can reduce based on the rules of the grammar

if len(self.stack) >= 2 and self.stack[-2:] == ['a', 'b']: # B → b, A → a, S → A B

self.stack = self.stack[:-2] # Remove 'a' and 'b'

self.stack.append('S') # Replace with the non-terminal 'S'

print("Reduced A B → S")

return True

return False

# Test the Bottom-Up Parser

input\_string = "ab"

parser = BottomUpParser(input\_string)

parser.parse()