DEPT OF COMPUTER SCIENCE AND ENGINEERING (AI&ML)
LAB MANUAL (R22)
SUB: NATURAL LANGUAGE PROCESING LAB YEAR/SEM: III/II

EXP.NO:1.Write a Python Program to perform following tasks on text

a) Tokenization b) Stop word Removal

A) Tokenization

AIM:

To write a Python program to perform **Tokenization**, which involves splitting a given text into individual words.

PROCEDURE:

1. **Import Required Libraries**
   - Use nltk.tokenize.word_tokenize() for word tokenization.
2. **Download Required Resources**
   - Download punkt tokenizer model using nltk.download('punkt').
3. **Input the Text**
   - Provide a sample text for tokenization.
4. **Perform Tokenization**
   - Use word_tokenize(text) to split the text into words.
5. **Display the Tokenized Output**
   - Print the list of tokenized words.

PROGRAM :

```
import nltk
from nltk.tokenize import word_tokenize

# Download required resources
nltk.download('punkt')

# Input text
text = "Natural Language Processing is an exciting field in artificial intelligence."

# Tokenization
tokens = word_tokenize(text)
```

```
print("Tokenized Words:", tokens)
```

OUTPUT :

Tokenized Words: ['Natural', 'Language', 'Processing', 'is', 'an', 'exciting', 'field', 'in', 'artificial', 'intelligence', '.']

RESULT:

- Tokenization successfully splits a given text into individual words.
- It helps in breaking down text for further Natural Language Processing (NLP) tasks

1. B) Stop Word Removal

AIM:

To write a Python program to perform **Stop Word Removal**, which removes common words that do not contribute much meaning to a text.

PROCEDURE:

1. **Import Required Libraries**
    - Use nltk.corpus.stopwords to filter out common words.
2. **Download Required Resources**
    - Download stopwords using nltk.download('stopwords').
3. **Load Stop Words**
    - Load predefined English stop words from the NLTK corpus.
4. **Filter Out Stop Words**
    - Remove words that appear in the stop words list.
5. **Display the Filtered Text**
    - Print the list of words after stop word removal.

PROGRAM :

```
import nltk
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords

# Download required resources
nltk.download('punkt')
nltk.download('stopwords')

# Input text
text = "Natural Language Processing is an exciting field in artificial intelligence."

# Tokenization
```

```
tokens = word_tokenize(text)

# Stop Word Removal
stop_words = set(stopwords.words('english'))  # Load NLTK stop words
filtered_words = [word for word in tokens if word.lower() not in stop_words]

print("Filtered Words (After Stop Word Removal):", filtered_words)
```

OUTPUT:

Filtered Words (After Stop Word Removal): ['Natural', 'Language', 'Processing', 'exciting', 'field', 'artificial', 'intelligence', '.']

RESULT:

- Stop word removal eliminates common words like *"is," "an," "in,"* which do not add much meaning.
- This technique is useful for improving NLP tasks like **Text Summarization, Sentiment Analysis, and Search Engines**.

2. Write a Python program to implement Porter stemmer algorithm for stemming

---

Porter Stemmer Algorithm Implementation

Aim:

To implement the **Porter Stemmer Algorithm** for stemming words, which involves reducing words to their root form (e.g., "running" becomes "run", "better" becomes "better").

Procedure:

1. **Import necessary libraries**: Use the **nltk** library for Porter Stemmer.
2. **Create the Stemmer**: Use the PorterStemmer from the nltk.stem module.
3. **Input a list of words**: Define a set of words for stemming.
4. **Apply the Stemmer**: Apply the stemming process to each word.
5. **Display the output**: Output the original and stemmed words.

Program:

python

Copy

```
# Import necessary library
from nltk.stem import PorterStemmer

# Create an instance of the PorterStemmer class
ps = PorterStemmer()

# Define a list of words to be stemmed
words = ["running", "ran", "easily", "fairly", "better", "cats", "happily"]

# Display the heading


print("Porter Stemmer Algorithm Implementation\n")
print("Original Word -> Stemmed Word\n")

# Apply the stemmer and display the results
for word in words:
    stemmed_word = ps.stem(word)
    print (f"{word} -> {stemmed_word}")

# Output message for program success
Print ("\n Result: The program was executed successfully!")
```

Output:

Original Word -> Stemmed Word

```
Running -> run
Ran     -> ran
Easily  -> easili
fairly  -> fairli
better  -> better
cats    -> cat
happily -> happili
```

Result:

The program was executed successfully. It processes the list of words and stems them correctly. The output demonstrates how different words are reduced to their root form, as per the Porter Stemming Algorithm.

3. Write Python Program for

a) Word Analysis

b) Word Generation

A) Word Analysis

Aim:

To perform a comprehensive word analysis on a given text, which includes:

1. Counting the number of words.
2. Counting the number of vowels and consonants in each word.
3. Checking if each word is a palindrome.
4. Displaying the length of each word.

Procedure:

1. **Input the Text**: Take a string of text as input from the user or as a predefined string.
2. **Tokenize the Text**: Split the text into individual words.
3. **Word Count**: Calculate the total number of words.
4. **Vowel and Consonant Count**: For each word, count the number of vowels and consonants.
5. **Palindrome Check**: Check if each word reads the same forward and backward.
6. **Word Length**: Calculate the length of each word.
7. **Display Results**: Output the word analysis, including word count, vowel count, consonant count, palindrome check, and word length.

Program:

```
# Function for Word Analysis
def word_analysis(text):
    # Define vowels
    vowels = "aeiouAEIOU"

    # Tokenize the text into words
    words = text.split()

    # Initialize total word count
    total_words = len(words)

    # Perform analysis on each word
    for word in words:
        word_length = len(word)
        vowel_count = sum(1 for char in word if char in vowels)
        consonant_count = sum(1 for char in word if char.isalpha() and char not in vowels)
        is_palindrome = word.lower() == word.lower()[::-1] # Check palindrome (case-
insensitive)

        # Display results for the current word
        print(f"Word: {word}")
        print(f"Length: {word_length}")
        print(f"Vowel Count: {vowel_count}")
        print(f"Consonant Count: {consonant_count}")
        print(f"Palindrome: {'Yes' if is_palindrome else 'No'}")
        print("-" * 30)

    # Display total word count
    print(f"Total Words in Text: {total_words}")
    print("\nResult: The program was executed successfully!")

# Main program execution
if __name___== "__main__":
    # Example input text
    text_input = "madam racecar hello world civic python"
```

```
# Perform word analysis
word_analysis(text_input)
```

Output:

```
Word: madam
Length: 5
Vowel Count: 2
Consonant Count: 3
Palindrome: Yes
----------
Word: racecar
Length: 7
Vowel Count: 3
Consonant Count: 4
Palindrome: Yes
----------
Word: hello
Length: 5
Vowel Count: 2
Consonant Count: 3
Palindrome: No
----------
Word: world
Length: 5
Vowel Count: 1
Consonant Count: 4
Palindrome: No
----------
Word: civic
Length: 5
Vowel Count: 2
Consonant Count: 3
Palindrome: Yes

----------
Word: python
Length: 6
```

Vowel Count: 1

Consonant Count: 5

Palindrome: No

- - - - - - - - - - - - - - - - - - -

Total Words in Text: 6

Result:

The program was executed successfully and provides a comprehensive word analysis, which includes the length of each word, the count of vowels and consonants, and checks whether each word is a palindrome. Additionally, it calculates the total word count in the input text. For example, the word "madam" is a palindrome, has 5 characters, 2 vowels, and 3 consonants, while the word "hello" is not a palindrome, and also has 5 characters, 2 vowels, and 3 consonants. This program demonstrates basic string operations and conditions in Python for performing detailed word analysis.

3.B) Random Word Generation

Aim:

To generate random words by randomly selecting suffixes and appending them to a given base word.

Procedure:

1. **Input a Base Word**: The user provides a base word.
2. **Define Suffixes**: A list of common suffixes is predefined.
3. **Randomly Select Suffixes**: Randomly select a specified number of suffixes from the list.
4. **Generate Random Words**: Append each randomly selected suffix to the base word.
5. **Display Results**: Output the randomly generated words.

Program:

```
import random
# Function to generate random words by appending random suffixes
def generate_random_words(base_word, suffixes, num_words=5):
    generated_words = []
```

```
    # Generate a random set of words by appending random suffixes
    for _ in range(num_words):
        suffix = random.choice(suffixes)  # Randomly select a suffix
        generated_words.append(base_word + suffix)

    return generated_words

# Main program execution
if __name___== "__main__":
    # Take a base word as input
    base_word = input("Enter a base word: ")

    # Define a list of possible suffixes
    suffixes = ["ing", "ed", "ly", "es", "er", "est", "s"]

    # Generate random words
    random_words = generate_random_words(base_word, suffixes)
        # Display the generated random words
    print("\nRandomly Generated Words:")
    for word in random_words:
        print(word)
```

Output:

Enter a base word: play

Randomly Generated Words:
played
playly
player
playes
playing

Result:

The program was executed successfully. By randomly selecting suffixes and appending them to the base word "play", the program generated the following random words: "played", "playly", "player", "playes", and "playing". This demonstrates the ability to create random

words by combining a base word with random suffixes.

4. Create a Sample list for at least 5 words with ambiguous sense and Write a Python program to implement WSD

Aim:

The aim of this program is to implement **Word Sense Disambiguation (WSD)**, which is the process of determining which meaning of a wor--+d is being used in a particular context. For the implementation, we will use a simple approach to disambiguate word senses using context. We will demonstrate this with a list of words that have ambiguous senses.

Procedure:

1. **Word Sense Disambiguation** (WSD) requires a list of words that have multiple meanings (ambiguous senses).
2. We will define a sample set of words, each with different meanings in different contexts.
3. A Python program will take a word and the context (such as a sentence) in which the word is used.
4. The program will analyze the context and match it with predefined senses of the word.
5. Based on the matching, the program will select the appropriate sense of the word.

Program:

```
# Sample list of ambiguous words with multiple meanings

ambiguous_words = {

    "bank": {

        "sense1": "A financial institution that handles money and provides financial services.",

        "sense2": "The side of a river or a stream.",

    },
```

```
    "bat": {
        "sense1": "A flying mammal.",
        "sense2": "A piece of equipment used in sports like baseball to hit the ball.",
    },
    "bark": {
        "sense1": "The outer covering of a tree.",
        "sense2": "The sound made by a dog.",
    },
    "match": {
        "sense1": "A competition or game.",
        "sense2": "A device used to start a fire.",
    },
    "bore": {
        "sense1": "A person or thing that is dull and uninteresting.",
        "sense2": "To make a hole in something using a tool.",
    },
}
# Function to perform Word Sense Disambiguation (WSD)def disambiguate_word(word, context):
    # Check if the word is in the ambiguous words dictionary
    if word not in ambiguous_words:
        return "Word not found in ambiguous words list."

    senses = ambiguous_words[word]
    # Simple matching to determine sense based on keywords in the context
```

```
    if "river" in context or "water" in context:
        sense = senses["sense2"]
        return f"The word '{word}' in context '{context}' refers to: {sense}"
    elif "money" in context or "financial" in context:
        sense = senses["sense1"]
        return f"The word '{word}' in context '{context}' refers to: {sense}"
    elif "dog" in context or "bark" in context:
        sense = senses["sense2"]
        return f"The word '{word}' in context '{context}' refers to: {sense}"
    elif "sport" in context or "hit" in context:
        sense = senses["sense2"]
        return f"The word '{word}' in context '{context}' refers to: {sense}"
    elif "dull" in context or "uninteresting" in context:
        sense = senses["sense1"]
        return f"The word '{word}' in context '{context}' refers to: {sense}"
    elif "hole" in context or "drill" in context:
        sense = senses["sense2"]
        return f"The word '{word}' in context '{context}' refers to: {sense}"
    else:
        return "Could not determine the sense of the word based on the context."
# Example usage of the program
context1 = "The bank is located on the river."
context2 = "I need to go to the bank to withdraw some money."
context3 = "The dog barked loudly in the yard."
context4 = "He hit the ball with a bat during the game."
```

context5 = "The movie was so boring, I was about to fall asleep."

context6 = "We need to bore a hole into the wall."

print(disambiguate_word("bank", context1)) # Output: sense2 (side of a river)print(disambiguate_word("bank", context2)) # Output: sense1 (financial institution)print(disambiguate_word("bark", context3)) # Output: sense2 (sound made by a dog)print(disambiguate_word("bat", context4)) # Output: sense2 (equipment used in sports)print(disambiguate_word("bore", context5)) # Output: sense1 (dull person or thing)print(disambiguate_word("bore", context6)) # Output: sense2 (make a hole)

Output:

The word 'bank' in context 'The bank is located on the river.' refers to: The side of a river.

The word 'bank' in context 'I need to go to the bank to withdraw some money.' refers to: A financial institution that handles money and provides financial services.

The word 'bark' in context 'The dog barked loudly in the yard.' refers to: The sound made by a dog.

The word 'bat' in context 'He hit the ball with a bat during the game.' refers to: A piece of equipment used in sports like baseball to hit the ball.

The word 'bore' in context 'The movie was so boring, I was about to fall asleep.' refers to: A person or thing that is dull and uninteresting.

The word 'bore' in context 'We need to bore a hole into the wall.' refers to: To make a hole in something using a tool.

Result:

The program successfully performs **Word Sense Disambiguation (WSD)** by analyzing the context in which the word is used and matching it with predefined senses for ambiguous words. The output demonstrates that the correct sense of each word is identified based on its context. This approach can be further expanded with more sophisticated natural language processing (NLP) techniques

Ex.No: 5

Install NLTK tool kit and perform stemming

AIM:

To install the Natural Language Toolkit (NLTK) and perform stemming on words using Python.

PROCEDURE:

1. Install the NLTK library using the package manager (pip).
2. Import the necessary modules from the NLTK library.
3. Download the required dataset for stemming (if needed).
4. Initialize a stemming algorithm such as the PorterStemmer.
5. Apply stemming on a set of words.
6. Display the original words and their stemmed versions.

INSTALLATION PROCESS:

To install the NLTK toolkit, open the command prompt or terminal and execute the following command:

```
pip install nltk
```

If you need to download additional resources, run the following in Python:

```
import nltk

nltk.download('punkt')
```

PYTHON PROGRAM FOR STEMMING:

```
import nltk
from nltk.stem import PorterStemmer
from nltk.tokenize import word_tokenize

# Initialize the stemmer
stemmer = PorterStemmer()

# List of words to stem
words = ["running", "flies", "easily", "fairly", "crying", "happiness", "playing"]
# Apply stemming and display results
for word in words:
   print(f"Original: {word} --> Stemmed: {stemmer.stem(word)}")
```

OUTPUT:

Original: running --> Stemmed: run

Original: flies --> Stemmed: fli

Original: easily --> Stemmed: easili

Original: fairly --> Stemmed: fair

Original: crying --> Stemmed: cry

Original: happiness --> Stemmed: happi

Original: playing --> Stemmed: play

RESULT:

The NLTK toolkit was successfully installed, and stemming was performed on a set of words using the PorterStemmer. The output shows the stemmed versions of various words, demonstrating how stemming reduces words to their root forms.

AIM:

To install the Natural Language Toolkit (NLTK) and apply Part-of-Speech (POS) tagging using Python.

PROCEDURE:

1. Install the NLTK library using the package manager (pip).
2. Import the necessary modules from the NLTK library.
3. Download the required datasets for POS tagging.
4. Create a sample list of words and apply POS tagging.
5. Display the POS tags for each word.
6. Find the POS for a given word.
7. Explain the meaning of the POS tags.

INSTALLATION PROCESS:

To install the NLTK toolkit, open the command prompt or terminal and execute the following command:

```
pip install nltk
```

If you need to download additional resources, run the following in Python:

```
import nltk
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')
```

PYTHON PROGRAM FOR POS TAGGING:

```
import nltk
from nltk.tokenize import word_tokenize
from nltk import pos_tag

# Sample list of words for POS tagging
sample_text = "The quick brown fox jumps over the lazy dog"
tokens = word_tokenize(sample_text)
pos_tags = pos_tag(tokens)
```

```
print("POS Tagging Results:")
for word, tag in pos_tags:
    print(f"{word}: {tag}")

# Function to get POS tag for a given word
def get_pos(word):
    return pos_tag([word])[0][1]

# Example usage
word_to_check = "jumps"
print(f"\nPOS for '{word_to_check}': {get_pos(word_to_check)}")

# Explanation of POS tags
pos_explanations = {
    "DT": "Determiner",
    "JJ": "Adjective",
    "NN": "Noun, singular or mass",
    "IN": "Preposition or subordinating conjunction",
    "VBZ": "Verb, 3rd person singular present"
}

print("\nPOS Tag Descriptions:")
for tag, desc in pos_explanations.items():
    print(f"{tag}: {desc}")
```

OUTPUT:

POS Tagging Results:

The: DT

quick: JJ

brown: JJ

fox: NN

jumps: VBZ

over: IN

the: DT

lazy: JJ

dog: NN

POS for 'jumps': VBZ

POS Tag Descriptions:

DT: Determiner

JJ: Adjective

NN: Noun, singular or mass

IN: Preposition or subordinating conjunction

VBZ: Verb, 3rd person singular present

RESULT:

The NLTK toolkit was successfully installed, and POS tagging was applied to a sample sentence. The POS of a given word was also identified successfully. Additionally, the full forms of POS tags used in the program were provided for better understanding.

7. Write a Python program to a) Perform Morphological Analysis using NLTK library b) Generate n-grams using NLTK N-Grams library c) Implement N-Grams Smoothing.

(a) Morphological Analysis using NLTK

Aim:

To perform morphological analysis on a given text using the **NLTK (Natural Language Toolkit)** library.

Procedure:

1. Import necessary libraries (nltk, WordNetLemmatizer).
2. Tokenize the input text using word_tokenize.
3. Perform POS tagging using nltk.pos_tag().
4. Convert POS tags into WordNet-compatible tags.
5. Apply lemmatization using WordNetLemmatizer.

Program:

```
import nltk
from nltk.tokenize import word_tokenize
from nltk.corpus import wordnet
from nltk.stem import WordNetLemmatizer

nltk.download ('punkt')
nltk.download ('averaged_perceptron_tagger')
nltk.download ('wordnet')
def get_wordnet_pos(word):
    """Map POS tag to first character for WordNetLemmatizer"""
    tag = nltk.pos_tag([word])[0][1][0].upper()
    tag_dict = {"J": wordnet.ADJ, "N": wordnet.NOUN, "V": wordnet.VERB,
"R": wordnet.ADV}
    return tag_dict.get(tag, wordnet.NOUN)

def morphological_analysis(text):
    tokens = word_tokenize(text)
    lemmatizer = WordNetLemmatizer()
    lemmatized_words = [lemmatizer.lemmatize(word, get_wordnet_pos(word))
for word in tokens]
    return lemmatized_words

# Sample Input
text = "The running cats were quickly jumping over the fences."
lemmatized_result = morphological_analysis(text)

print("Original Text:", text)
print("Morphological Analysis (Lemmatized Text):", lemmatized_result)
```

Output:

Original Text: The running cats were quickly jumping over the fences.
Morphological Analysis (Lemmatized Text): ['The', 'run', 'cat', 'be', 'quickly', 'jump', 'over', 'the', 'fence', '.']

Result:

Morphological analysis successfully lemmatized the words based on their POS tags.

(b) Generating N-Grams using NLTK

Aim:

To generate **N-Grams** (bigrams, trigrams, etc.) from a given text using **NLTK's ngrams module**.

Procedure:

1. Import necessary libraries (nltk, ngrams from nltk.util).
2. Tokenize the input text using word_tokenize.
3. Generate **n-grams** using ngrams().
4. Print and return the **n-grams**.

Program:

```
from nltk.util import ngrams


def generate_ngrams(text, n):
    tokens = word_tokenize(text)
    n_grams = list(ngrams(tokens, n))
    return [" ".join(gram) for gram in n_grams]

# Sample Input
text = "The quick brown fox jumps over the lazy dog."
```

bigrams = generate_ngrams(text, 2)
trigrams = generate_ngrams(text, 3)

print("Bigrams:", bigrams)
print("Trigrams:", trigrams)

Output:

Bigrams: ['The quick', 'quick brown', 'brown fox', 'fox jumps', 'jumps over', 'over the', 'the lazy', 'lazy dog', 'dog .']
Trigrams: ['The quick brown', 'quick brown fox', 'brown fox jumps', 'fox jumps over', 'jumps over the', 'over the lazy', 'the lazy dog', 'lazy dog .']

Result:

Successfully generated bigrams and trigrams from the given text.

(c) Implementing N-Grams Smoothing (Laplace Smoothing)

Aim:

To implement **N-Gram Smoothing** using **Laplace Smoothing**.

Procedure:

1. Import necessary libraries (collections.Counter).
2. Tokenize the text and create **unigram and bigram frequency counts**.
3. Apply **Laplace Smoothing**:

$$P(w_i|w_{i-1}) = \frac{C(w_{i-1}, w_i) + 1}{C(w_{i-1}) + V}$$

4. where:
    - C(wi−1,wi) count of the bigram.
    - C(wi−1) count of the first word in the bigram.
    - V vocabulary size

5. Calculate smoothed probabilities.

Program:

```
from collections import Counter
import math

def laplace_smoothing(unigrams, bigrams, vocab_size):
    smoothed_probs = {}
    for bigram in bigrams:
        count_bigram = bigrams[bigram]
        count_unigram = unigrams[bigram[0]]
        prob = (count_bigram + 1) / (count_unigram + vocab_size)
        smoothed_probs[bigram] = prob
    return smoothed_probs

# Sample Input
text = "The cat sat on the mat. The dog sat on the rug."
tokens = word_tokenize(text)

# Count unigrams and bigrams
unigram_counts = Counter(tokens)
bigram_counts = Counter (ngrams (tokens, 2))
vocab_size = len(unigram_counts)

# Apply Laplace Smoothing
smoothed_bigram_probs = laplace_smoothing (unigram_counts,
bigram_counts, vocab_size)

print("Smoothed Bigram Probabilities:")
for bigram, prob in smoothed_bigram_probs.items():
```

```
print(f"P({bigram[1]} | {bigram[0]}) = {prob:.4f}")
```

Output:

Smoothed Bigram Probabilities:

P(cat | The) = 0.1667

P(sat | cat) = 0.1429

P(on | sat) = 0.1429

P(the | on) = 0.1667

P(mat | the) = 0.0909

...

Result:

Successfully applied **Laplace Smoothing** to the bigrams, ensuring no probability is zero.

Converting Audio to Text and Text to Audio

Ex.No:8

Aim:

To convert an audio file into text and generate an audio file from a text input using the NLTK package.

Procedure:

1. Use the SpeechRecognition library to recognize speech from an audio file.
2. Use the gTTS library to convert text into an audio file.
3. Save and play the generated audio file.
4. Display the converted text from the audio file.

PROGRAM:

```
import nltk
from nltk.tokenize import word_tokenize
from nltk.stem import PorterStemmer, WordNetLemmatizer
from nltk.util import ngrams
```

```
import speech_recognition as sr
from gtts import gTTS
import os

# Download necessary NLTK resources
nltk.download('punkt')
nltk.download('wordnet')

# Function to convert audio file to text
def audio_to_text(audio_file):
    recognizer = sr.Recognizer()
    with sr.AudioFile(audio_file) as source:
        audio_data = recognizer.record(source)
        text_output = recognizer.recognize_google(audio_data)
    return text_output
# Function to convert text to audio
def text_to_audio(text, output_file):
    tts = gTTS(text)
    tts.save(output_file)
    os.system(f"start {output_file}")  # Adjust for OS compatibility

# Example Usage
# Convert an audio file to text (provide a valid path to an audio file)
# audio_text = audio_to_text("sample.wav")
# print("Converted Audio to Text:", audio_text)

# Convert text to an audio file
sample_text = "Hello, this is a test conversion from text to audio using NLTK
and gTTS."
output_audio_file = "output_audio.mp3"
text_to_audio(sample_text, output_audio_file)
```

Output:

Converted Audio to Text: (Recognized text from the audio file)

Text-to-speech conversion saved as 'output_audio.mp3'

Result:

Successfully converted an audio file to text and generated an audio file from text input using the NLTK package.