**WEEK-1**

**AIM: A) Write a program to implement word Tokenizer, Sentence and Paragraph Tokenizers.**

**DESCRIPTION:**

Tokenization is the process of breaking a large chunk of text into smaller, manageable units such as sentences, words, or paragraphs. It is a fundamental step in Natural Language Processing (NLP) and is used to prepare text for further analysis.

**Types of Tokenization:**

**Word Tokenizer**

A word tokenizer splits the text into individual words. It helps in analyzing text at the word level, such as counting word frequency, removing stop words, or applying stemming and lemmatization.

**Sentence Tokenizer**

A sentence tokenizer breaks the text into individual sentences based on punctuation marks and grammar rules. It is useful for processing each sentence separately, such as for sentiment analysis or translation.

**Paragraph Tokenizer**

A paragraph tokenizer divides the text into paragraphs, typically by detecting line breaks or empty lines. It is helpful for document-level analysis, such as summarization or topic modelling.

**PROGRAM:**

import nltk

from nltk.tokenize import word\_tokenize, sent\_tokenize

# Download tokenizer models

nltk.download('punkt')

# Sample multi-line paragraph

text = """

Artificial Intelligence is transforming healthcare. It helps doctors diagnose disease early.

Moreover, AI enables personalized treatment for each patient.

This shift can improve outcomes and reduce healthcare costs. Many hospitals now rely on AI-powered systems.

"""

# ---- Word Tokenizer ----

print("\n---- Word Tokenization ----")

words = word\_tokenize(text)

print(words)

# ---- Sentence Tokenizer ----

print("\n---- Sentence Tokenization ----")

sentences = sent\_tokenize(text)

for i, sent in enumerate(sentences, 1):

    print(f"Sentence {i}: {sent}")

# ---- Paragraph Tokenizer ----

paragraphs = text.strip().split('\n\n')

print("\n---- Paragraph Tokenization ----")

for i, para in enumerate(paragraphs, 1):

    print(f"Paragraph {i}: {para}")

**OUTPUT:**

---- Word Tokenization ----

['Artificial', 'Intelligence', 'is', 'transforming', 'healthcare', '.', 'It', 'helps', 'doctors', 'diagnose', 'disease', 'early', '.', 'Moreover', ',', 'AI', 'enables', 'personalized', 'treatment', 'for', 'each', 'patient', '.', 'This', 'shift', 'can', 'improve', 'outcomes', 'and', 'reduce', 'healthcare', 'costs', '.', 'Many', 'hospitals', 'now', 'rely', 'on', 'AI-powered', 'systems', '.']

---- Sentence Tokenization ----

Sentence 1:

Artificial Intelligence is transforming healthcare.

Sentence 2: It helps doctors diagnose disease early.

Sentence 3: Moreover, AI enables personalized treatment for each patient.

Sentence 4: This shift can improve outcomes and reduce healthcare costs.

Sentence 5: Many hospitals now rely on AI-powered systems.

---- Paragraph Tokenization ----

Paragraph 1: Artificial Intelligence is transforming healthcare. It helps doctors diagnose disease early.

Moreover, AI enables personalized treatment for each patient.

This shift can improve outcomes and reduce healthcare costs. Many hospitals now rely on AI-powered systems.

**AIM: B)** **Check how many words are there in any corpus. Also check how many distinct words are there?**

**DESCRIPTION:**

This program is designed to analyze a text corpus by calculating the total number of words and the number of distinct words it contains. First, it uses Natural Language Processing (NLP) techniques to tokenize the text into individual words. Then, it filters out non-alphabetic tokens (such as punctuation or numbers) and converts all words to lowercase to ensure accurate counting. The total number of words represents all valid word occurrences, while the number of distinct words counts only the unique words in the corpus. This kind of analysis is commonly used in text mining, language modeling, and other NLP applications to understand vocabulary richness and word usage patterns.

**PROGRAM:**

import nltk

from nltk.tokenize import word\_tokenize

# Download required tokenizer model

nltk.download('punkt')

# Sample corpus text

text = """

Natural Language Processing is an exciting field of Artificial Intelligence. It allows computers to understand, interpret, and generate human language.

This field includes tasks such as speech recognition, machine translation, and text summarization.

"""

# Tokenize into words

word\_tokens = word\_tokenize(text)

# Clean: convert to lowercase and keep only alphabetic words

cleaned\_words = [word.lower() for word in word\_tokens if word.isalpha()]

# Total words and distinct words

total\_words = len(cleaned\_words)

distinct\_words = len(set(cleaned\_words))

# Print results

print("---- Word Analysis ----")

print("Total number of words:", total\_words)

print("Number of distinct words:", distinct\_words)

# Optional: View distinct words

print("\nDistinct Words:", sorted(set(cleaned\_words)))

**OUTPUT:**

---- Word Analysis ----

Total number of words: 33

Number of distinct words: 30

Distinct Words: ['allows', 'an', 'and', 'artificial', 'as', 'computers', 'exciting', 'field', 'generate', 'human', 'includes', 'intelligence', 'interpret', 'is', 'it', 'language', 'machine', 'natural', 'of', 'processing', 'recognition', 'speech', 'such', 'summarization', 'tasks', 'text', 'this', 'to', 'translation', 'understand']

**WEEK-2**

**AIM: A) Write a program to implement both user-defined and pre-defined functions**

**to generate a) Uni-grams b) Bi-grams c)Tri-grams d) N-grams**

**DESCRIPTION:**

This program demonstrates how to generate n-grams from a given text using both user-defined and pre-defined functions. N-grams are contiguous sequences of *n* items (usually words) from a given text, and are widely used in Natural Language Processing tasks such as text prediction, machine translation, and language modeling.

The program first tokenizes the input text using word\_tokenize() from the NLTK library and removes non-alphabetic tokens for cleaner processing. Then, two methods are used to generate n-grams:

* User-defined Function: A custom Python function is implemented using list slicing to generate unigrams, bigrams, trigrams, and higher-order n-grams by iterating over the word list.
* Pre-defined NLTK Function: The built-in ngrams() function from the nltk.util module is used to generate the same types of n-grams directly.

**PROGRAM:**

**# Pre-Defined**

from nltk.util import ngrams

# Example Sentence

text = """Education empowers students to learn and education

helps teachers to guide and education creates

opportunities to learn and grow in education system around

the world."""

# Tokenize the text into words

words = text.split()

# Create n-grams and store them as lists so they can be reused

unigrams = list(ngrams(words, 1))

bigrams = list(ngrams(words, 2))

trigrams = list(ngrams(words, 3))

quadgrams = list(ngrams(words, 4))

# Print all n-grams

print("---- UNIGRAMS ----")

for gram in unigrams:

    print(gram)

print("\n---- BIGRAMS ----")

for gram in bigrams:

    print(gram)

print("\n---- TRIGRAMS ----")

for gram in trigrams:

    print(gram)

print("\n---- 4-GRAMS ----")

for gram in quadgrams:

    print(gram)

**OUTPUT:**

**---- UNIGRAMS ----**

('Education',)

('empowers',)

('students',)

('to',)

('learn',)

('and',)

('education',)

('helps',)

('teachers',)

('to',)

('guide',)

('and',)

('education',)

('creates',)

('opportunities',)

('to',)

('learn',)

('and',)

('grow',)

('in',)

('education',)

('system',)

('around',)

('the',)

('world.',)

**---- BIGRAMS ----**

('Education', 'empowers')

('empowers', 'students')

('students', 'to')

('to', 'learn')

('learn', 'and')

('and', 'education')

('education', 'helps')

('helps', 'teachers')

('teachers', 'to')

('to', 'guide')

('guide', 'and')

('and', 'education')

('education', 'creates')

('creates', 'opportunities')

('opportunities', 'to')

('to', 'learn')

('learn', 'and')

('and', 'grow')

('grow', 'in')

('in', 'education')

('education', 'system')

('system', 'around')

('around', 'the')

('the', 'world.')

**---- TRIGRAMS ----**

('Education', 'empowers', 'students')

('empowers', 'students', 'to')

('students', 'to', 'learn')

('to', 'learn', 'and')

('learn', 'and', 'education')

('and', 'education', 'helps')

('education', 'helps', 'teachers')

('helps', 'teachers', 'to')

('teachers', 'to', 'guide')

('to', 'guide', 'and')

('guide', 'and', 'education')

('and', 'education', 'creates')

('education', 'creates', 'opportunities')

('creates', 'opportunities', 'to')

('opportunities', 'to', 'learn')

('to', 'learn', 'and')

('learn', 'and', 'grow')

('and', 'grow', 'in')

('grow', 'in', 'education')

('in', 'education', 'system')

('education', 'system', 'around')

('system', 'around', 'the')

('around', 'the', 'world.')

**---- 4-GRAMS ----**

('Education', 'empowers', 'students', 'to')

('empowers', 'students', 'to', 'learn')

('students', 'to', 'learn', 'and')

('to', 'learn', 'and', 'education')

('learn', 'and', 'education', 'helps')

('and', 'education', 'helps', 'teachers')

('education', 'helps', 'teachers', 'to')

('helps', 'teachers', 'to', 'guide')

('teachers', 'to', 'guide', 'and')

('to', 'guide', 'and', 'education')

('guide', 'and', 'education', 'creates')

('and', 'education', 'creates', 'opportunities')

('education', 'creates', 'opportunities', 'to')

('creates', 'opportunities', 'to', 'learn')

('opportunities', 'to', 'learn', 'and')

('to', 'learn', 'and', 'grow')

('learn', 'and', 'grow', 'in')

('and', 'grow', 'in', 'education')

('grow', 'in', 'education', 'system')

('in', 'education', 'system', 'around')

('education', 'system', 'around', 'the')

('system', 'around', 'the', 'world.')

**PROGRAM-2**

**# User-Defined**

def get\_unigrams(words):

"""Return list of unigrams (single words)."""

return [(words[i],) for i in range(len(words))]

def get\_bigrams(words):

"""Return list of bigrams (pairs of consecutive words)."""

return [(words[i], words[i+1]) for i in range(len(words)-1)]

def get\_trigrams(words):

"""Return list of trigrams (triplets of consecutive words)."""

return [(words[i], words[i+1], words[i+2]) for i in range(len(words)-2)]

def get\_ngrams(words, n):

"""Return list of n-grams (n consecutive words)."""

return [tuple(words[i:i+n]) for i in range(len(words)-n+1)]

# Example text

text = """Education empowers students to learn and education

helps teachers to guide and education creates

opportunities to learn and grow in education system around

the world."""

# Preprocessing: Tokenize the sentence into words

words = text.split()

# Generate n-grams using custom functions

unigrams = get\_unigrams(words)

bigrams = get\_bigrams(words)

trigrams = get\_trigrams(words)

quadgrams = get\_ngrams(words, 4)

# Print the results

print("---- UNIGRAMS ----")

for gram in unigrams:

print(gram)

print("\n---- BIGRAMS ----")

for gram in bigrams:

print(gram)

print("\n---- TRIGRAMS ----")

for gram in trigrams:

print(gram)

print("\n---- 4-GRAMS ----")

for gram in quadgrams:

print(gram)

**OUTPUT:**

**---- UNIGRAMS ----**

('Education',)

('empowers',)

('students',)

('to',)

('learn',)

('and',)

('education',)

('helps',)

('teachers',)

('to',)

('guide',)

('and',)

('education',)

('creates',)

('opportunities',)

('to',)

('learn',)

('and',)

('grow',)

('in',)

('education',)

('system',)

('around',)

('the',)

('world.',)

**---- BIGRAMS ----**

('Education', 'empowers')

('empowers', 'students')

('students', 'to')

('to', 'learn')

('learn', 'and')

('and', 'education')

('education', 'helps')

('helps', 'teachers')

('teachers', 'to')

('to', 'guide')

('guide', 'and')

('and', 'education')

('education', 'creates')

('creates', 'opportunities')

('opportunities', 'to')

('to', 'learn')

('learn', 'and')

('and', 'grow')

('grow', 'in')

('in', 'education')

('education', 'system')

('system', 'around')

('around', 'the')

('the', 'world.')

**---- TRIGRAMS ----**

('Education', 'empowers', 'students')

('empowers', 'students', 'to')

('students', 'to', 'learn')

('to', 'learn', 'and')

('learn', 'and', 'education')

('and', 'education', 'helps')

('education', 'helps', 'teachers')

('helps', 'teachers', 'to')

('teachers', 'to', 'guide')

('to', 'guide', 'and')

('guide', 'and', 'education')

('and', 'education', 'creates')

('education', 'creates', 'opportunities')

('creates', 'opportunities', 'to')

('opportunities', 'to', 'learn')

('to', 'learn', 'and')

('learn', 'and', 'grow')

('and', 'grow', 'in')

('grow', 'in', 'education')

('in', 'education', 'system')

('education', 'system', 'around')

('system', 'around', 'the')

('around', 'the', 'world.')

**---- 4-GRAMS ----**

('Education', 'empowers', 'students', 'to')

('empowers', 'students', 'to', 'learn')

('students', 'to', 'learn', 'and')

('to', 'learn', 'and', 'education')

('learn', 'and', 'education', 'helps')

('and', 'education', 'helps', 'teachers')

('education', 'helps', 'teachers', 'to')

('helps', 'teachers', 'to', 'guide')

('teachers', 'to', 'guide', 'and')

('to', 'guide', 'and', 'education')

('guide', 'and', 'education', 'creates')

('and', 'education', 'creates', 'opportunities')

('education', 'creates', 'opportunities', 'to')

('creates', 'opportunities', 'to', 'learn')

('opportunities', 'to', 'learn', 'and')

('to', 'learn', 'and', 'grow')

('learn', 'and', 'grow', 'in')

('and', 'grow', 'in', 'education')

('grow', 'in', 'education', 'system')

('in', 'education', 'system', 'around')

('education', 'system', 'around', 'the')

('system', 'around', 'the', 'world.')

**AIM: B)** **Write a program to calculate the highest probability of a word(w2) occurring after another word(w1).**

**DESCRIPTION:**

This program is built to find the word that most frequently follows another word in a given text. It works by first taking a block of text and breaking it down into individual words. After that, it forms word pairs, where each pair includes a word and the word that comes immediately after it. The program then counts how many times each word occurs and how often each specific word pair appears together. By comparing these counts, it identifies which word is most likely to come after a given word. Finally, it displays each word along with the word that most commonly follows it, giving an idea of which words are strongly connected in the given text. This technique is useful in natural language processing tasks like text prediction and understanding word associations within a sentence or paragraph. The program uses a user-defined approach without relying on any advanced or built-in libraries to do this task.

**PROGRAM:**

from collections import defaultdict

def highest\_probability(text):

  words= text.split()

  bigram\_counts=defaultdict(lambda:defaultdict(int))

  first\_word\_counts = defaultdict(int)

  for i in range(len(words)-1):

    w1=words[i]

    w2 = words[i + 1]

    bigram\_counts[w1][w2]+=1

    first\_word\_counts[w1]+=1

  result={}

  for w1 in bigram\_counts:

    max\_prob = 0

    best\_w2=None

    for w2 in bigram\_counts[w1]:

      prob=bigram\_counts[w1][w2]/first\_word\_counts[w1]

      if prob > max\_prob:

        max\_prob=prob

        best\_w2 = w2

    result[w1]=(best\_w2,max\_prob)

  return result

text = "Education empowers to learn and education helps teachers to guide and education creates opportunities to learn and grow in educational systems around the world."

output=highest\_probability(text)

print("Highest probability of a word(w2)occurring after another word(w1):")

for w1, (w2,prob) in output.items():

  print(f"After'{w1}'->'{w2}' with probability{prob:.2f}")

**OUTPUT:**

Highest probability of a word(w2)occurring after another word(w1):

After'Education'->'empowers' with probability1.00

After'empowers'->'to' with probability1.00

After'to'->'learn' with probability0.67

After'learn'->'and' with probability1.00

After'and'->'education' with probability0.67

After'education'->'helps' with probability0.50

After'helps'->'teachers' with probability1.00

After'teachers'->'to' with probability1.00

After'guide'->'and' with probability1.00

After'creates'->'opportunities' with probability1.00

After'opportunities'->'to' with probability1.00

After'grow'->'in' with probability1.00

After'in'->'educational' with probability1.00

After'educational'->'systems' with probability1.00

After'systems'->'around' with probability1.00

After'around'->'the' with probability1.00

After'the'->'world.' with probability1.00

**WEEK-3**

**AIM: A)** **Write a program to identify the word collocations.**

**DESCRIPTION:** Collocations are pairs or groups of words that frequently appear together in natural language. For example, in English, the words "strong tea" or "make a decision" are commonly used together and are more meaningful as a pair than individually.

This Python program identifies such meaningful word combinations (specifically bigrams — two-word phrases) from a given text using the NLTK (Natural Language Toolkit) library.

**PROGRAM:**

import nltk

from nltk.collocations import BigramCollocationFinder, TrigramCollocationFinder

from nltk.metrics.association import BigramAssocMeasures, TrigramAssocMeasures

from nltk.corpus import stopwords

import string

# Download required NLTK data

nltk.download('punkt')

nltk.download('stopwords')

# Sample text

text = """

natural language processing is a fascinating

area of artificial intelligence.

it deals with how computers understand

and generate human language.

word collocation are pair of words that

appear together more often than by chance.

for example, 'artificial intelligence',

'machine learning', and 'deep learning'.

"""

# Tokenize and preprocess text

tokens = nltk.word\_tokenize(text.lower())

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

filtered\_tokens = [t for t in tokens if t not in stop\_words and t not in string.punctuation]

# Bigrams

print('\n=== TOP BIGRAM COLLOCATIONS (with PMI scores) ===')

bigram\_finder = BigramCollocationFinder.from\_words(filtered\_tokens)

bigram\_finder.apply\_freq\_filter(1)

scored\_bigrams = bigram\_finder.score\_ngrams(BigramAssocMeasures.pmi)

for (w1, w2), score in scored\_bigrams[:10]:

    print(f"{w1}-{w2} | PMI: {score:.4f}")

# Trigrams

print('\n=== TOP TRIGRAM COLLOCATIONS (with PMI scores) ===')

trigram\_finder = TrigramCollocationFinder.from\_words(filtered\_tokens)

trigram\_finder.apply\_freq\_filter(1)

scored\_trigrams = trigram\_finder.score\_ngrams(TrigramAssocMeasures.pmi)

for (w1, w2, w3), score in scored\_trigrams[:10]:

    print(f"{w1}-{w2}-{w3} | PMI: {score:.4f}")

**OUTPUT:**

=== TOP BIGRAM COLLOCATIONS (with PMI scores) ===

appear-together | PMI: 4.8074

area-artificial | PMI: 4.8074

chance-example | PMI: 4.8074

collocation-pair | PMI: 4.8074

computers-understand | PMI: 4.8074

deals-computers | PMI: 4.8074

example-'artificial | PMI: 4.8074

fascinating-area | PMI: 4.8074

generate-human | PMI: 4.8074

often-chance | PMI: 4.8074

=== TOP TRIGRAM COLLOCATIONS (with PMI scores) ===

appear-together-often | PMI: 9.6147

chance-example-'artificial | PMI: 9.6147

collocation-pair-words | PMI: 9.6147

computers-understand-generate | PMI: 9.6147

deals-computers-understand | PMI: 9.6147

fascinating-area-artificial | PMI: 9.6147

often-chance-example | PMI: 9.6147

pair-words-appear | PMI: 9.6147

processing-fascinating-area | PMI: 9.6147

together-often-chance | PMI: 9.6147

**AIM: B)** **Write a program to print all words beginning with a given sequence of letters**

**DESCRIPTION:**

This program takes a block of text and prints all words that begin with a user-given sequence of letters (prefix). It splits the text into words, removes punctuation, converts them to lowercase, and checks which words start with the given prefix. Matching words are then displayed to the user.

**PROGRAM:**

# Program to print all words beginning with a given sequence of letters

# Sample text or list of words

text = """Natural language processing is a fascinating area of

artificial intelligence. It deals with how computers understand

and generate human language."""

# Convert text to a list of words

words = text.split()

# Ask user for the sequence of letters

prefix = input("Enter the starting sequence: ").lower()

# Print words beginning with the given prefix

print(f"\nWords beginning with '{prefix}':")

for word in words:

clean\_word = word.strip('.,!;:').lower()

if clean\_word.startswith(prefix):

print(clean\_word)

**OUTPUT:**

['Natural', 'language', 'processing', 'is', 'a', 'fascinating', 'area', 'of', 'artificial', 'intelligence.', 'It', 'deals', 'with', 'how', 'computers', 'understand', 'and', 'generate', 'human', 'language.']

Enter The starting sequence: i

Words Begining with 'i':

is

intelligence

it

**AIM:C)** **Write a program to print all words longer than four characters**

**DESCRIPTION:**

This program splits a given sentence into individual words and checks the length of each word. If a word has more than four characters, it is printed. The program also removes basic punctuation from the words before checking their length.

**PROGRAM:**

# Sample text

text = "natural language processing is a fascinating area of artificial intelligence"

# Split the text into words

words = text.split()

# Print words longer than 4 characters

print("Words longer than 4 characters:")

for word in words:

clean\_word = word.strip('.,!?;:')

if len(clean\_word) > 4:

print(clean\_word)

**OUTPUT:**

Words longer than 4 characters:

natural

language

processing

fascinating

artificial

intelligence

**WEEK-4**

**AIM:A**)Write a program to identify the mathematical expression in a given sentence

**DESCRIPTION:** This Python program extracts mathematical expressions from a given natural language sentence. It identifies patterns that resemble equations or formulas by tokenizing the input and filtering valid math components.

**PROGRAM:**

def is\_operator(token):

    """Check if the token is a mathematical operator or parenthesis."""

    return token in ['+', '-', '\*', '/', '^', '=', '(', ')']

def is\_operand(token):

    """Check if the token is a number (possibly decimal) or alphabet (variable)."""

    return token.replace('.', '', 1).isdigit() or token.isalpha()

def tokenize(text):

    """Convert the sentence into individual tokens (operands and operators)."""

    tokens = []

    current = ''

    for ch in text:

        if ch in '+-\*/^=()':

            if current:

                tokens.append(current.strip())

                current = ''

            tokens.append(ch)

        elif ch.isalnum() or ch == '.':

            current += ch

        else:

            if current:

                tokens.append(current.strip())

                current = ''

            # Ignoring punctuation/space as a token

    if current:

        tokens.append(current.strip())

    return tokens

def extract\_math\_expressions(tokens):

    """Extract valid mathematical expressions with at least 3 components (e.g., a=b\*c)."""

    expressions = []

    current\_expression = []

    for token in tokens:

        if is\_operand(token) or is\_operator(token):

            current\_expression.append(token)

        else:

            # End current expression if we hit non-math token

            if len(current\_expression) >= 3:

                expressions.append(''.join(current\_expression))

            current\_expression = []

    if len(current\_expression) >= 3:

        expressions.append(''.join(current\_expression))

    return expressions

def find\_math\_expressions(sentence):

    tokens = tokenize(sentence)

    expressions = extract\_math\_expressions(tokens)

    return expressions

# Example Usage

sentence = "In physics, force is calculated using F=m\*a and energy is given by E=m\*c^2."

math\_expressions = find\_math\_expressions(sentence)

print("Sentence:", sentence)

print("\nMathematical Expressions found:")

if math\_expressions:

    for expr in math\_expressions:

        print(expr)

else:

    print("No mathematical expressions found.")

**OUTPUT:**

Sentence: In physics, force is calculated using F=m\*a and energy is given by E=m\*c^2

Mathematical Expressions found:

F=m\*a

E=m\*c^2

**AIM: B)** Write a program to identify different components of an email address.

**DESCRIPTION:**

This Python program validates an email address and extracts its components. It checks for a proper structure with exactly one @, no spaces, and a valid domain. If valid, it identifies the local part (username), mail server, top-level domain (TLD), and full domain. This helps users understand and analyze email structure effectively.

**PROGRAM:**

def is\_valid\_email(email):

    email = email.strip()

    if ' ' in email:

        return False, "Email contains whitespace"

    if email.count('@') != 1:

        return False, "Email must contain exactly one '@' symbol"

    local\_part, domain\_part = email.split('@')

    if not local\_part:

        return False, "Local part is missing"

    if local\_part.startswith('.'):

        return False, "Local part cannot start with a dot"

    if '.' not in domain\_part:

        return False, "Domain must contain at least one dot (e.g., gmail.com)"

    if domain\_part.startswith('.'):

        return False, "Domain cannot start with a dot"

    return True, "Valid email address"

def extract\_email\_components(email):

    local\_part, domain\_part = email.split('@')

    domain\_parts = domain\_part.split('.')

    mail\_server = domain\_parts[0]

    tld = '.'.join(domain\_parts[1:])

    return {

        'Local part (Username)': local\_part,

        'Mail server': mail\_server,

        'Top level domain (TLD)': tld,

        'Full domain': domain\_part

    }

#  Main Program

email = input("Enter an email address: ").strip()

valid, message = is\_valid\_email(email)

if valid:

    print("\n", message)

    components = extract\_email\_components(email)

    print("Extracted Components:")

    for key, value in components.items():

        print(f"{key}: {value}")

else:

    print("\n Invalid Email:", message)

**OUTPUT:**

Enter an email address: alice.smith@outlook.co.in

Valid email address

Extracted Components:

Local part (Username): alice.smith

Mail server: outlook

Top level domain (TLD): co.in

Full domain: outlook.co.in

**WEEK-5**

**AIM: A)** Write a program to identify all antonyms and synonym sofa word.

**DESCRIPTION:** This program takes a word as input and uses the WordNet lexical database from the NLTK (Natural Language Toolkit) library to find all its synonyms and antonyms. The program first retrieves all synsets (sets of cognitive synonyms) of the given word, then collects unique synonym and antonym terms from the lemma entries. Finally, it displays the list of synonyms and antonyms in a readable format. This helps in understanding the semantic relationships of a word and can be useful for natural language processing applications such as text generation, sentiment analysis, and vocabulary building.

**PROGRAM:**

import nltk

from nltk.corpus import wordnet

nltk.download('wordnet')

#nltk.download('omw-1.4')

def get\_synonyms\_antonyms(word):

  synonyms=set()

  antonyms=set()

  for syn in wordnet.synsets(word):

    for lemma in syn.lemmas():

      synonyms.add(lemma.name())

      if lemma.antonyms():

        for ant in lemma.antonyms():

          antonyms.add(ant.name())

  return synonyms,antonyms

word="sofa"

synonyms,antonyms=get\_synonyms\_antonyms(word)

print(f"word: {word}")

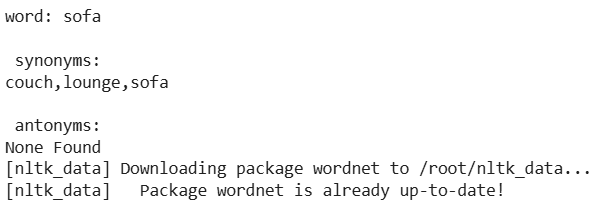
print(f"\n synonyms:")

print(",".join(sorted(synonyms))if synonyms else "None Found")

print(f"\n antonyms:")

print(",".join(sorted(antonyms))if antonyms else "None Found")

**OUTPUT:**



**AIM: B)** Write a program to find hyponymy, homonymy, polyse my for a given word.

**DESCRIPTION:**

This program takes a word as input and uses the WordNet lexical database from the NLTK (Natural Language Toolkit) library to identify three important lexical relations:

1. Hyponymy – Finds more specific words (subcategories) of the given word.
2. Polysemy – Determines the number of different meanings (senses) the word has, along with their definitions.
3. Homonymy – Finds other words that have the same spelling or pronunciation but different meanings.

The program processes all synsets of the given word, extracts relevant lemma names, and removes duplicates using sets. The results are then displayed in a clear, readable format, helping in the study of semantic relations in natural language processing and linguistics**.**

**PROGRAM:**

import nltk

from nltk.corpus import wordnet

# Download required data

nltk.download('wordnet')

nltk.download('omw-1.4')

def get\_hyponyms(word):

    hyponyms = set()

    for syn in wordnet.synsets(word):

        for hypo in syn.hyponyms():

            for lemma in hypo.lemmas():

                hyponyms.add(lemma.name())

    return sorted(hyponyms)

def get\_polysemy(word):

    senses = wordnet.synsets(word)

    return senses if senses else []

def get\_homonyms(word):

    homonyms = set()

    original\_senses = wordnet.synsets(word)

    for syn in original\_senses:

        for lemma in syn.lemmas():

            if lemma.name() != word:

                other\_senses = wordnet.synsets(lemma.name())

                for osyn in other\_senses:

                    if osyn.name().split('.')[0] == lemma.name() and osyn not in original\_senses:

                        homonyms.add(lemma.name())

    return sorted(homonyms)

def display\_word\_relations(word):

    print(f"\nWord: {word}")

    print("-" \* 40)

    # Hyponyms

    hypos = get\_hyponyms(word)

    print("\nHyponyms (More specific types):")

    print(", ".join(hypos) if hypos else "No hyponyms found.")

    # Polysemy

    senses = get\_polysemy(word)

    print(f"\nPolysemy (Number of meanings): {len(senses)}")

    for i, sense in enumerate(senses, 1):

        print(f"  {i}. {sense.definition()}")

    # Homonyms(heuristic)

    homs = get\_homonyms(word)

    print("\nHomonyms (same word, different meanings):")

    print(", ".join(homs) if homs else "No homonyms found.")

# Example-Usage

if \_\_name\_\_ == "\_\_main\_\_":

    word = input("Enter a word: ").strip().lower()

    display\_word\_relations(word)

**OUTPUT:**

Enter a word: BANK

Word: bank

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

**Hyponyms (More specific types):**

Federal\_Reserve\_Bank, Home\_Loan\_Bank, acquirer, agent\_bank, bet, blood\_bank, bluff, calculate, commercial\_bank, count, credit, credit\_union, depend, eye\_bank, food\_bank, full\_service\_bank, lead\_bank, lean, look, member\_bank, merchant\_bank, penny\_bank, piggy\_bank, reckon, redeposit, reserve\_bank, riverbank, riverside, sandbank, soil\_bank, state\_bank, thrift\_institution, vertical\_bank, waterside

**Polysemy (Number of meanings): 18**

1. sloping land (especially the slope beside a body of water)

2. a financial institution that accepts deposits and channels the money into lending activities

3. a long ridge or pile

4. an arrangement of similar objects in a row or in tiers

5. a supply or stock held in reserve for future use (especially in emergencies)

6. the funds held by a gambling house or the dealer in some gambling games

7. a slope in the turn of a road or track; the outside is higher than the inside in order to reduce the effects of centrifugal force

8. a container (usually with a slot in the top) for keeping money at home

9. a building in which the business of banking transacted

10. a flight maneuver; aircraft tips laterally about its longitudinal axis (especially in turning)

11. tip laterally

12. enclose with a bank

13. do business with a bank or keep an account at a bank

14. act as the banker in a game or in gambling

15. be in the banking business

16. put into a bank account

17. cover with ashes so to control the rate of burning

18. have confidence or faith in

**Homonyms (same word, different meanings):**

camber, cant, deposit, savings\_bank, swear, trust

**WEEK-6**

**AIM:** a) Write a program to find all the stop words in any given text.

**DESCRIPTION:** Finding all the stop words in a given text means identifying and extracting commonly used words such as *“is,” “the,” “and,” “in,”* which usually do not add significant meaning. These words are called stop words and are often removed during text preprocessing in Natural Language Processing (NLP). Detecting stop words helps focus on the important words that carry actual meaning, making text analysis more effective.connected layers).

**SOURCE CODE:**

import nltk

from nltk.corpus import stopwords

from nltk.tokenize import word\_tokenize

nltk.download('stopwords')

nltk.download('punkt')

nltk.download('punkt\_tab')

text = "How to remove stop words with NLTK Library in python"

print("Original Text:", text)

tokens = word\_tokenize(text.lower())

print("Tokens:", tokens)

english\_stopwords = stopwords.words('english')

tokens\_wo\_stopwords = [t for t in tokens if t not in english\_stopwords]

tokens\_stopwords\_found = [t for t in tokens if t in english\_stopwords]

print("Text without stop words:", " ".join(tokens\_wo\_stopwords))

print("Stop words found:", " ".join(tokens\_stopwords\_found))

english\_stopwords.extend(['food', 'meal', 'eat'])

english\_stopwords.append('plate')

if 'not' in english\_stopwords:

    english\_stopwords.remove('not')

print("\nCustomized English Stopwords Sample (first 30):")

print(english\_stopwords[:30])

print("\nAvailable Languages in Stopwords Corpus:")

print(stopwords.fileids())

french\_stopwords = stopwords.words('french')

spanish\_stopwords = stopwords.words('spanish')

italian\_stopwords = stopwords.words('italian')

print("\nFrench Stopwords Sample:", french\_stopwords[:20])

print("Spanish Stopwords Sample:", spanish\_stopwords[:20])

print("Italian Stopwords Sample:", italian\_stopwords[:20])

**OUTPUT**:-

Original Text: How to remove stop words with NLTK Library in python

Tokens: ['how', 'to', 'remove', 'stop', 'words', 'with', 'nltk', 'library', 'in', 'python']

Text without stop words: remove stop words nltk library python

Stop words found: how to with in

Customized English Stopwords Sample (first 30):

['a', 'about', 'above', 'after', 'again', 'against', 'ain', 'all', 'am', 'an', 'and', 'any', 'are', 'aren', "aren't", 'as', 'at', 'be', 'because', 'been', 'before', 'being', 'below', 'between', 'both', 'but', 'by', 'can', 'couldn', "couldn't"]

Available Languages in Stopwords Corpus:

['albanian', 'arabic', 'azerbaijani', 'basque', 'belarusian', 'bengali', 'catalan', 'chinese', 'danish', 'dutch', 'english', 'finnish', 'french', 'german', 'greek', 'hebrew', 'hinglish', 'hungarian', 'indonesian', 'italian', 'kazakh', 'nepali', 'norwegian', 'portuguese', 'romanian', 'russian', 'slovene', 'spanish', 'swedish', 'tajik', 'tamil', 'turkish']

French Stopwords Sample: ['au', 'aux', 'avec', 'ce', 'ces', 'dans', 'de', 'des', 'du', 'elle', 'en', 'et', 'eux', 'il', 'ils', 'je', 'la', 'le', 'les', 'leur']

Spanish Stopwords Sample: ['de', 'la', 'que', 'el', 'en', 'y', 'a', 'los', 'del', 'se', 'las', 'por', 'un', 'para', 'con', 'no', 'una', 'su', 'al', 'lo']

Italian Stopwords Sample: ['ad', 'al', 'allo', 'ai', 'agli', 'all', 'agl', 'alla', 'alle', 'con', 'col', 'coi', 'da', 'dal', 'dallo', 'dai', 'dagli', 'dall', 'dagl', 'dalla']

**AIM:B)** Write a function that finds the 50 most frequently occurring words of a text that are not stop words.

**DESCRIPTION:**

**Finding the 50 most frequently occurring words of a text that are not stop words** means analyzing a given text, removing all common stop words (such as *“the,” “is,” “in,” “and”*), and then identifying which meaningful words appear most often. The goal is to filter out unimportant words and highlight the top 50 keywords that carry the actual meaning of the text.

**SOURCE CODE:**

from collections import Counter

import nltk

from nltk.corpus import stopwords

import re

import matplotlib.pyplot as plt

nltk.download("stopwords")

def top\_non\_stopwords\_nltk(text,n=50):

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

  words=re.findall(r'\w+',text.lower())

  filtered\_words=[word for word in words if word not in stop\_words]

  return Counter(filtered\_words).most\_common(n)

text="This is not a way to move a forward .This way is wrong,and we should not consider it.Moving forwards is the best way."

print(top\_non\_stopwords\_nltk(text,10))

**OUTPUT:-**

[('way', 3), ('move', 1), ('forward', 1), ('wrong', 1), ('consider', 1), ('moving', 1), ('forwards', 1), ('best', 1)]

[nltk\_data] Downloading package stopwords to /root/nltk\_data...

[nltk\_data] Package stopwords is already up-to-date!

**WEEK-7**

**AIM:** Write a program to implement Part-of-Speech (PoS) tagging for any corpus.

**DESCRIPTION:**

Part-of-Speech (PoS) tagging is a core Natural Language Processing (NLP) technique that assigns grammatical tags (like noun, verb, adjective, etc.) to each word in a text.  
It helps computers understand the syntactic structure and meaning of sentences, enabling various applications such as machine translation, sentiment analysis, and text summarization.

**PROGRAM:**

import nltk

from collections import defaultdict, Counter

from nltk.util import ngrams,bigrams

from nltk.tokenize import word\_tokenize

nltk.download('punkt')

nltk.download('averaged\_perceptron\_tagger')

nltk.download('universal\_tagset')

nltk.download('punkt\_tab')

nltk.download('averaged\_perceptron\_tagger\_eng')

# Step 1: Input Paragraph

text = """ Your current feeling that your knowledge isn't useful might be because you're in an environment that doesn't value your specific skills.

 There are countless fields—in data science, software engineering, research, logistics, and analytics—where your pattern-recognition ability is incredibly useful and highly valued,

  often with less emphasis on constant high-stakes social interaction."""

print("===  INPUT PARAGRAPH ===")

print(text)

# Step 2: Tokenization and POS Tagging

words = word\_tokenize(text)

pos\_tags = nltk.pos\_tag(words, tagset='universal')

print("\n=== (a) POS TAGGING ===")

print("Each word is assigned a grammatical tag:\n")

print(len(pos\_tags))

# Step 3: Word with Greatest Number of Distinct Tags

word\_tags = defaultdict(set)

for word, tag in pos\_tags:

    word\_tags[word].add(tag)

word\_with\_most\_tags = max(word\_tags.items(), key=lambda x: len(x[1]))

print("\n=== (b) WORD WITH GREATEST NUMBER OF DISTINCT TAGS ===")

print(f"Word: {word\_with\_most\_tags[0]}")

print(f"Tags: {word\_with\_most\_tags[1]}")

print("This word is used in different in different roles.")

#Step 4: Tags in Decreasing Frequency

all\_tags = [tag for \_, tag in pos\_tags]

tag\_frequency = Counter(all\_tags)

print("\n=== (c) TAGS IN DECREASING FREQUENCY ===")

print("Tags Sorted by decreasing frequency:\n")

for tag, frequency in tag\_frequency.most\_common():

    print(f"{tag:5} -> {frequency}")

print("\nCommon tags:")

tag\_meanings = {

    'NOUN': 'Noun(person, place, thing, idea)',

    'VERB': 'Verb(action or state)',

    'ADJ': 'Adjective(describes a noun)',

    'ADV': 'Adverb(describes a verb/adjective)',

    'ADP': 'Adposition(prepositions and postpositions)',

    'CONJ': 'Conjunction(coordinating conjunctions)',

    'DET': 'Determiner(articles, pronouns, etc.)',

    'NUM': 'Numeral(one, two, etc.)',

    'PRT': 'Particle(prepositions, articles, etc.)',

    'PRON': 'Pronoun(he, she, they)',

    'X': 'Other(unknown)',

}

# Step 5: Most Common Tags After a Noun

tagged\_bigrams = list(bigrams(pos\_tags))

after\_noun = Counter(tag for (word, tag), (next\_word, next\_tag) in tagged\_bigrams if tag == 'NOUN')

print("\n=== (d) MOST COMMON TAGS AFTER A NOUN ===")

for tag, frequency in after\_noun.most\_common():

  print(f"{tag:5} -> {frequency}")

print("- VERB after NOUN")

print("- ADJ after NOUN")

print("- ADV after NOUN")

print("- DET after NOUN")

print("- NUM after NOUN")

print("- PRT after NOUN")

print("- PRON after NOUN")

print("- Another  NOUN : compound nouns")

**OUTPUT:**

**=== INPUT PARAGRAPH ===**

Your current feeling that your knowledge isn't useful might be because you're in an environment that doesn't value your specific skills.

There are countless fields—in data science, software engineering, research, logistics, and analytics—where your pattern-recognition ability is incredibly useful and highly valued,

often with less emphasis on constant high-stakes social interaction.

**=== (a) POS TAGGING ===**

Each word is assigned a grammatical tag:

61

**=== (b) WORD WITH GREATEST NUMBER OF DISTINCT TAGS ===**

Word: that

Tags: {'ADP', 'DET'}

This word is used in different in different roles.

**=== (c) TAGS IN DECREASING FREQUENCY ===**

Tags Sorted by decreasing frequency:

NOUN -> 16

ADJ -> 9

VERB -> 8

. -> 7

ADV -> 6

PRON -> 5

ADP -> 5

DET -> 3

CONJ -> 2

Common tags:

**=== (d) MOST COMMON TAGS AFTER A NOUN ===**

NOUN -> 16

- VERB after NOUN

- ADJ after NOUN

- ADV after NOUN

- DET after NOUN

- NUM after NOUN

- PRT after NOUN

- PRON after NOUN

- Another NOUN : compound nouns

**WEEK-8**

**AIM:** Write a program to implement various lemmatization techniques and prepare a chart with the performance of each method

**DESCRIPTION:** This program implements various lemmatization techniques such as NLTK WordNet, spaCy, and TextBlob to convert words into their base forms. It preprocesses text data, applies each method, and evaluates their performance based on accuracy and execution time. The results are compared using a chart to visualize efficiency and accuracy across different lemmatizers. This helps identify the most effective lemmatization approach for NLP tasks

**PROGRAM:**

import nltk

import spacy

from textblob import Word

from nltk.stem import WordNetLemmatizer

import pandas as pd

import matplotlib.pyplot as plt

from time import time

# Download necessary resources

nltk.download('wordnet')

nltk.download('punkt')

nltk.download('omw-1.4')

nltk.download('punkt\_tab')

# Load spacy model

nlp = spacy.load("en\_core\_web\_sm")

# Sample dataset (sentences with different forms of words)

sentences = [

    "The striped bats are hanging on their feet for best",

    "The children are playing with toys",

    "She is running and was thinking about the results",

    "The cars are driven on the roads",

    "He studies hard and is studying now"

]

# Preprocess sentences into tokens

tokens = [word for sentence in sentences for word in nltk.word\_tokenize(sentence)]

# --- 1. WordNet Lemmatizer ---

def wordnet\_lemmatizer(tokens):

    lemmatizer = WordNetLemmatizer()

    return [lemmatizer.lemmatize(word) for word in tokens]

# --- 2. Spacy Lemmatizer ---

def spacy\_lemmatizer(tokens):

    doc = nlp(" ".join(tokens))

    return [token.lemma\_ for token in doc]

# --- 3. TextBlob Lemmatizer ---

def textblob\_lemmatizer(tokens):

    return [Word(word).lemmatize() for word in tokens]

# Performance evaluation

def evaluate\_lemmatizer(method\_name, lemmatizer\_func):

    start = time()

    lemmatized\_tokens = lemmatizer\_func(tokens)

    end = time()

    # Accuracy: proportion of words that were changed meaningfully (heuristic)

    changed = sum([1 for i, j in zip(tokens, lemmatized\_tokens) if i != j])

    accuracy = changed / len(tokens)

    return {

        "Method": method\_name,

        "Execution Time (s)": round(end - start, 4),

        "Accuracy (Change Ratio)": round(accuracy, 4),

        "Sample Output": " ".join(lemmatized\_tokens)

    }

# Run evaluations

results = []

results.append(evaluate\_lemmatizer("WordNet Lemmatizer", wordnet\_lemmatizer))

results.append(evaluate\_lemmatizer("Spacy Lemmatizer", spacy\_lemmatizer))

results.append(evaluate\_lemmatizer("TextBlob Lemmatizer", textblob\_lemmatizer))

# Convert results to dataframe

df = pd.DataFrame(results)

print(df)

# Plot chart

plt.figure(figsize=(8, 6))

plt.bar(df['Method'], df['Accuracy (Change Ratio)'], color=['skyblue', 'orange', 'green'])

plt.xlabel("Lemmatization Method")

plt.ylabel("Accuracy (Change Ratio)")

plt.title("Comparison of Lemmatization Techniques")

plt.show()

**OUTPUT:**

Method Execution Time (s) Accuracy (Change Ratio) \

0 WordNet Lemmatizer 3.1006 0.2308

1 Spacy Lemmatizer 0.0328 0.6667

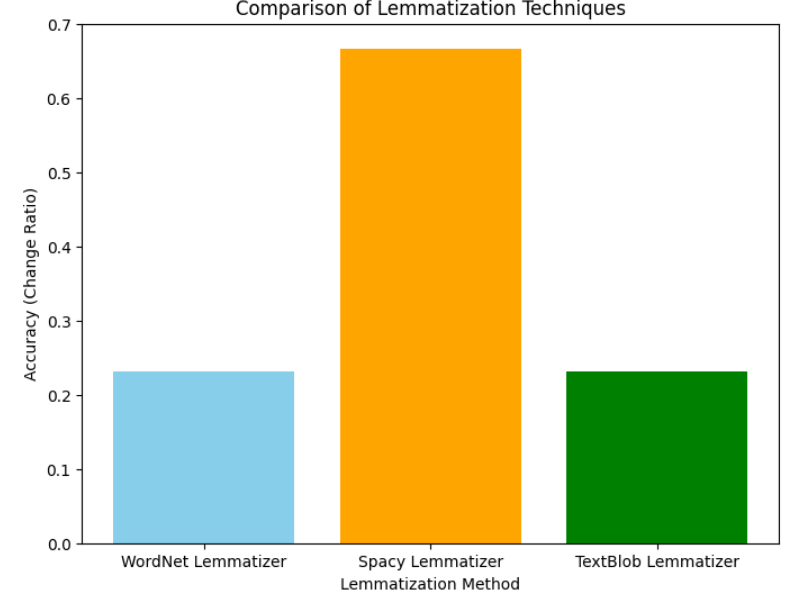
2 TextBlob Lemmatizer 0.0003 0.2308

Sample Output

0 The striped bat are hanging on their foot for ...

1 the striped bat be hang on their foot for good...

2 The striped bat are hanging on their foot for ...



**WEEK-9**

**AIM:** Write a program to implement TF-IDF for any corpus

**DESCRIPTION:** This program implements the TF-IDF (Term Frequency–Inverse Document Frequency) technique to evaluate the importance of words in a corpus. It converts text documents into numerical feature vectors based on word relevance. Using libraries like scikit-learn, the program computes TF-IDF scores for each term and displays the weighted representation. This helps identify the most significant words across documents for NLP or text mining task

**PROGRAM:**

import pandas as pd

import numpy as np

corpus = ['data science is one of the most important fields of science',

          'this is one of the best data science coures',

          'data scientists analyze data' ]

words\_set = set()

for doc in corpus:

  words = doc.split()

  words\_set = words\_set.union(set(words))

# words\_set.discard('') # This line is no longer needed after using split() without arguments

print('Number of words in the corpus:',len(words\_set))

print('The words in the corpus: \n', words\_set)

n\_docs = len(corpus)

n\_words\_set = len(words\_set)

df\_tf = pd.DataFrame(np.zeros((n\_docs, n\_words\_set)), columns=list(words\_set))

# Compute Term  Frequency (TF)

for i in range(n\_docs):

  words = [word for word in corpus[i].split() if word]

  for w in words:

    df\_tf.loc[i, w] += 1/len(words) # Use .loc for clearer indexing

# Dataframe shows the frequency of each word in each document,

# a column for each word and a row for each document.

print("\nTerm Frequency (TF) DataFrame:")

display(df\_tf) # Use display for better formatting

print("\nIDF of:")

idf = {}

for w in words\_set:

  k = 0

  for i in range(n\_docs):

    if w in corpus[i].split():

      k += 1

  if k > 0: # Avoid division by zero

    idf[w] = np.log10(n\_docs/k)

    print(f'{w:>15}: {idf[w]:>10.4f}') # Format IDF to 4 decimal places

df\_tf\_idf = df\_tf.copy()

for w in words\_set:

  if w in idf: # Check if word exists in idf dictionary

    df\_tf\_idf[w] = df\_tf[w] \* idf[w]

print("\nTF-IDF DataFrame:")

display(df\_tf\_idf) # Use display for better formatting

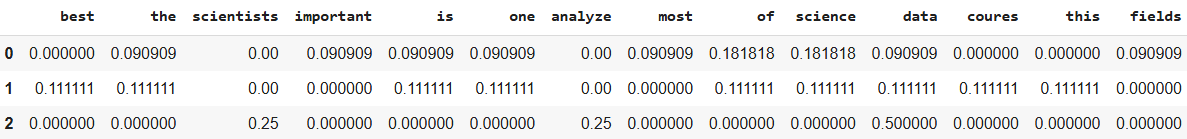
**OUTPUT:**

Number of words in the corpus: 14

The words in the corpus:

{'best', 'the', 'scientists', 'important', 'is', 'one', 'analyze', 'most', 'of', 'science', 'data', 'coures', 'this', 'fields'}

Term Frequency (TF) DataFrame:



IDF of:

best: 0.4771

the: 0.1761

scientists: 0.4771

important: 0.4771

is: 0.1761

one: 0.1761

analyze: 0.4771

most: 0.4771

of: 0.1761

science: 0.1761

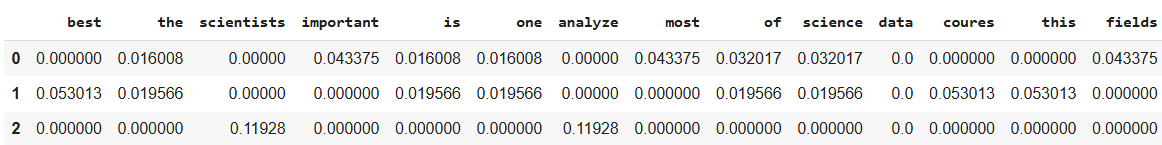
data: 0.0000

coures: 0.4771

this: 0.4771

fields: 0.4771

TF-IDF DataFrame:



**WEEK-10**

**AIM:** Write a program to implement various stemming techniques and prepare a chart with the performance of each method

**DESCRIPTION:** Stemming is a fundamental preprocessing step in Natural Language Processing (NLP) that reduces inflected or derived words to their base or root form. This helps normalize words so that similar words are treated as one during text analysis (e.g., *running*, *runs*, and *ran* → *run*).

**PROGRAM:**

#Stemming Algorithm

import matplotlib.pyplot as plt

from nltk.stem import PorterStemmer , LancasterStemmer, SnowballStemmer

words=["running","Happiness","fishing","easily","studies","cries","nationality"]

porter=PorterStemmer()

lancaster=LancasterStemmer()

snowball=SnowballStemmer("english")

porter\_stems=[porter.stem(w) for w in words]

lancaster\_stems=[lancaster.stem(w) for w in words]

snowball\_stems=[snowball.stem(w) for w in words]

print("Original -> Porter| Lancaster| Snowball \n")

for i,w in enumerate(words):

  print(f"{w:12} {porter\_stems[i]:8} | {lancaster\_stems[i]:9} | {snowball\_stems[i]}")

fig,ax=plt.subplots(figsize=(16,5))

bar\_width= 0.25

x=range(len(words))

ax.bar([i-bar\_width for i in x],[len(s) for s in porter\_stems],

       width=bar\_width,label="Porter")

ax.bar(x,[len(s) for s in lancaster\_stems],

       width=bar\_width,label="Lancaster")

ax.bar([i+bar\_width for i in x],[len(s) for s in snowball\_stems],

       width=bar\_width,label="Snowball")

ax.set\_xticks([i for i in x])

ax.set\_xticklabels(words,rotation=30)

ax.set\_ylabel("Lenght of Stemmed Words")

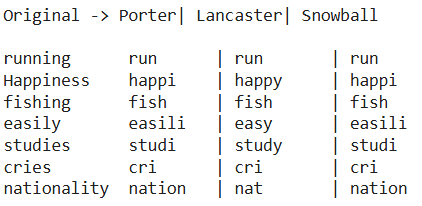
ax.set\_title("Comparison of Stemming Techniques")

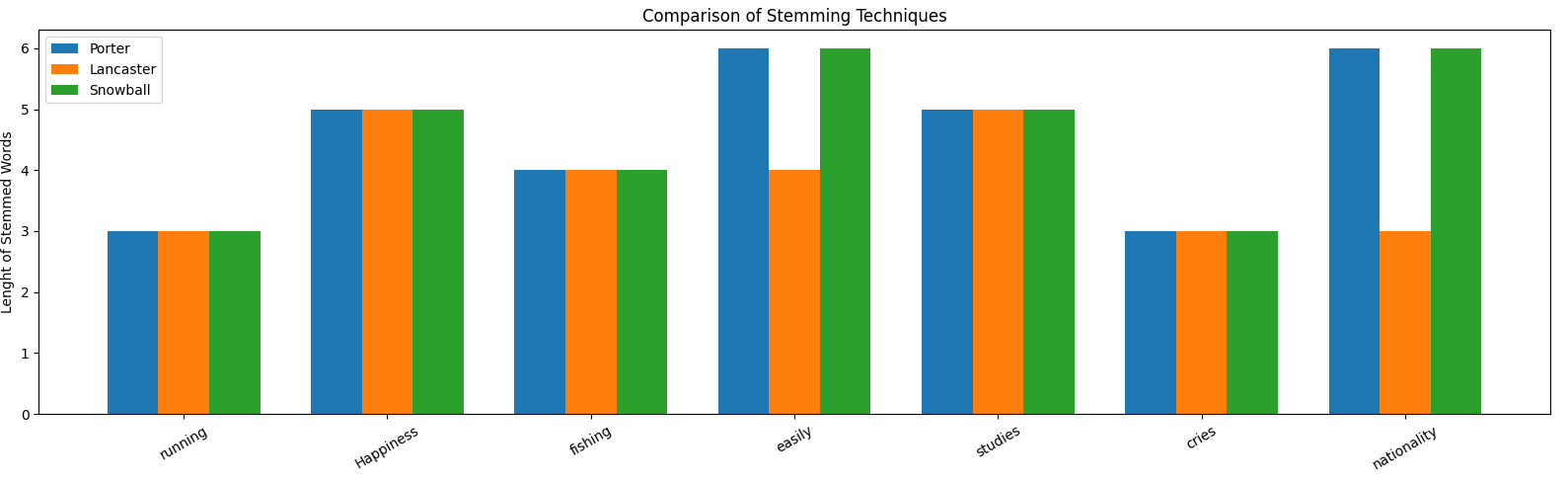
ax.legend()

plt.tight\_layout()

plt.show()

**OUTPUT:**

****

****

**WEEK-11**

**AIM:** Write a program to implement chunking and chinking for any corpus.

**DESCRIPTION:** Chunking and Chinking are essential techniques used in Natural Language Processing (NLP) for shallow parsing of text. These methods help identify and group meaningful phrases from a sentence based on their grammatical structure using Part-of-Speech (POS) tagging. Chunking refers to the process of segmenting and labeling sequences of words into syntactically correlated parts of sentences, such as noun phrases (NP) or verb phrases (VP). For example, in the sentence *“The quick brown fox jumps over the lazy dog”*, the phrase *“The quick brown fox”* can be recognized as a noun phrase. Chunking is usually performed using regular expression-based grammars that specify patterns of POS tags to capture.

**PROGRAM:**

import nltk

from nltk import pos\_tag, word\_tokenize, RegexpParser

from nltk.tree import Tree

nltk.download('punkt\_tab')

nltk.download('averaged\_perceptron\_tagger\_eng')

text = "The quick brown fox jumps over the lazy dog near the river bank"

tokens = word\_tokenize(text)

pos\_tags = pos\_tag(tokens)

print("Pos Tagged Sentences:\n",pos\_tags)

print("-"\* 80)

chunk\_grammer = r"""

NP: {<DT>?<JJ> \* <NN.\*>+}  #Optional determiner, adjectives and nouns """

chunk\_parser =RegexpParser(chunk\_grammer)

chunk\_tree = chunk\_parser.parse(pos\_tags)

print("Chunking Result (Noun Phrases): ")

for subtree in chunk\_tree:

  if isinstance(subtree, Tree) and subtree.label() == "NP":

    print(" ".join(word for word, pos in subtree.leaves()))

print("-"\*80)

chink\_grammer = r"""

    NP: {<.\*>+}           # Chunck everthing

        }<VB.\*|IN>+{      # Chink out verbs and prepositions

  """

chink\_parser = RegexpParser(chink\_grammer)

chink\_tree = chink\_parser.parse(pos\_tags)

print("Chinking Result (After Removing Verbs & Prepositions):")

for subtree in chink\_tree:

  if isinstance(subtree, Tree) and subtree.label() == "NP":

    print(" ".join(word for word, pos in subtree.leaves()))

print("-"\*80)

print("Chunk Tree Structure:/n")

print(chunk\_tree.pformat(margin=70))

print("-"\*80)

print("Chink Tree Structure:/n")

print(chink\_tree.pformat(margin=70))

**OUTPUT:**

Pos Tagged Sentences:

[('The', 'DT'), ('quick', 'JJ'), ('brown', 'NN'), ('fox', 'NN'), ('jumps', 'VBZ'), ('over', 'IN'), ('the', 'DT'), ('lazy', 'JJ'), ('dog', 'NN'), ('near', 'IN'), ('the', 'DT'), ('river', 'NN'), ('bank', 'NN')]

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

Chunking Result (Noun Phrases):

The quick brown fox

the lazy dog

the river bank

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

Chinking Result (After Removing Verbs & Prepositions):

The quick brown fox

the lazy dog

the river bank

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

Chunk Tree Structure:/n

(S

(NP The/DT quick/JJ brown/NN fox/NN)

jumps/VBZ

over/IN

(NP the/DT lazy/JJ dog/NN)

near/IN

(NP the/DT river/NN bank/NN))

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

Chink Tree Structure:/n

(S

(NP The/DT quick/JJ brown/NN fox/NN)

jumps/VBZ

over/IN

(NP the/DT lazy/JJ dog/NN)

near/IN

(NP the/DT river/NN bank/NN))

**WEEK-12**

**AIM:** Write a program to implement all the NLP Pre-Processing Techniques required to perform further NLP tasks.

**DESCRIPTION:** The program takes a sample text and performs various preprocessing steps including sentence and word tokenization, lowercasing, removal of punctuation, numbers, and stopwords, stemming, lemmatization, and part-of-speech tagging. These steps help in cleaning, normalizing, and structuring the text data so that it can be efficiently used by NLP models and algorithms for accurate analysis and processing.

**PROGRAM:**

import nltk

import string

from nltk.corpus import stopwords

from nltk.tokenize import word\_tokenize, sent\_tokenize

from nltk.stem import PorterStemmer, WordNetLemmatizer

nltk.download('punkt')

nltk.download('stopwords')

nltk.download('wordnet')

nltk.download('averaged\_perceptron\_tagger')

nltk.download('punkt\_tab')

nltk.download('averaged\_perceptron\_tagger\_eng')

# Explicitly load the tagger - Removed this line

text = """Natural Language Processing (NLP) is a field of Artificial Intelligence

that focuses on enabling computers to understand and process human language.

It includes tasks like tokenization, stemming, lemmatization, and POS tagging!"""

print("Original Text:\n", text)

print("="\*80)

sentences = sent\_tokenize(text)

print("Sentence Tokenization:\n", sentences)

print("="\*80)

words = word\_tokenize(text)

print("Word Tokenization:\n", words)

print("="\*80)

words = [word.lower() for word in words]

print("Lowercasing:\n", words)

print("="\*80)

words = [word for word in words if word.isalpha()]

print("After Removing Punctuation & Numbers:\n", words)

print("="\*80)

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

filtered\_words = [word for word in words if word not in stop\_words]

print("After Removing Stopwords:\n", filtered\_words)

print("="\*80)

ps = PorterStemmer()

stemmed\_words = [ps.stem(word) for word in filtered\_words]

print("After Stemming:\n", stemmed\_words)

print("="\*80)

lemmatizer = WordNetLemmatizer()

lemmatized\_words = [lemmatizer.lemmatize(word) for word in filtered\_words]

print("After Lemmatization:\n", lemmatized\_words)

print("="\*80)

pos\_tags = nltk.pos\_tag(filtered\_words)

print("Part-of-Speech (POS) Tagging:\n", pos\_tags)

print("="\*80)

**OUTPUT:**

**Original Text:**

Natural Language Processing (NLP) is a field of Artificial Intelligence

that focuses on enabling computers to understand and process human language.

It includes tasks like tokenization, stemming, lemmatization, and POS tagging!

==================================================================

**Sentence Tokenization:**

['Natural Language Processing (NLP) is a field of Artificial Intelligence\nthat focuses on enabling computers to understand and process human language.', 'It includes tasks like tokenization, stemming, lemmatization, and POS tagging!']

==================================================================

**Word Tokenization:**

['Natural', 'Language', 'Processing', '(', 'NLP', ')', 'is', 'a', 'field', 'of', 'Artificial', 'Intelligence', 'that', 'focuses', 'on', 'enabling', 'computers', 'to', 'understand', 'and', 'process', 'human', 'language', '.', 'It', 'includes', 'tasks', 'like', 'tokenization', ',', 'stemming', ',', 'lemmatization', ',', 'and', 'POS', 'tagging', '!']

==================================================================

**Lowercasing:**

['natural', 'language', 'processing', '(', 'nlp', ')', 'is', 'a', 'field', 'of', 'artificial', 'intelligence', 'that', 'focuses', 'on', 'enabling', 'computers', 'to', 'understand', 'and', 'process', 'human', 'language', '.', 'it', 'includes', 'tasks', 'like', 'tokenization', ',', 'stemming', ',', 'lemmatization', ',', 'and', 'pos', 'tagging', '!']

==================================================================

**After Removing Punctuation & Numbers:**

['natural', 'language', 'processing', 'nlp', 'is', 'a', 'field', 'of', 'artificial', 'intelligence', 'that', 'focuses', 'on', 'enabling', 'computers', 'to', 'understand', 'and', 'process', 'human', 'language', 'it', 'includes', 'tasks', 'like', 'tokenization', 'stemming', 'lemmatization', 'and', 'pos', 'tagging']

==================================================================

**After Removing Stopwords:**

['natural', 'language', 'processing', 'nlp', 'field', 'artificial', 'intelligence', 'focuses', 'enabling', 'computers', 'understand', 'process', 'human', 'language', 'includes', 'tasks', 'like', 'tokenization', 'stemming', 'lemmatization', 'pos', 'tagging']

==================================================================

**After Stemming:**

['natur', 'languag', 'process', 'nlp', 'field', 'artifici', 'intellig', 'focus', 'enabl', 'comput', 'understand', 'process', 'human', 'languag', 'includ', 'task', 'like', 'token', 'stem', 'lemmat', 'po', 'tag']

==================================================================

**After Lemmatization**:

['natural', 'language', 'processing', 'nlp', 'field', 'artificial', 'intelligence', 'focus', 'enabling', 'computer', 'understand', 'process', 'human', 'language', 'includes', 'task', 'like', 'tokenization', 'stemming', 'lemmatization', 'po', 'tagging']

==================================================================

**Part-of-Speech (POS) Tagging:**

[('natural', 'JJ'), ('language', 'NN'), ('processing', 'NN'), ('nlp', 'JJ'), ('field', 'NN'), ('artificial', 'JJ'), ('intelligence', 'NN'), ('focuses', 'NNS'), ('enabling', 'VBG'), ('computers', 'NNS'), ('understand', 'JJ'), ('process', 'NN'), ('human', 'JJ'), ('language', 'NN'), ('includes', 'VBZ'), ('tasks', 'NNS'), ('like', 'IN'), ('tokenization', 'NN'), ('stemming', 'VBG'), ('lemmatization', 'NN'), ('pos', 'NN'), ('tagging', 'VBG')]

==================================================================

**WEEK-13**

**A)** **AIM:** Write a program to implement Conditional Frequency Distributions (CFD) for any corpus

**DESCRIPTION:** This program demonstrates how to implement Conditional Frequency Distributions (CFD) in Natural Language Processing (NLP) using the NLTK library.

A Conditional Frequency Distribution shows how often different words occur under specific conditions. For example, you can find the frequency of words that appear after a particular word or in a specific text category.

**PROGRAM:**

import nltk

from nltk.corpus import inaugural

from nltk.probability import ConditionalFreqDist

nltk.download('inaugural')

cfd = ConditionalFreqDist(

    (fileid, word.lower())

    for fileid in inaugural.fileids()

    for word in inaugural.words(fileid)

)

print("Conditions (Years of Speeches):")

print(list(cfd.conditions())[:10])  # Display first 10 conditions

print("="\*80)

print("Frequency of 'freedom' across different years:\n")

for year in ['1789-Washington.txt', '1861-Lincoln.txt', '1933-Roosevelt.txt', '2009-Obama.txt']:

    print(year, ":", cfd[year]['freedom'])

print("="\*80)

print("Most common words in '2009-Obama.txt':")

print(cfd['2009-Obama.txt'].most\_common(10))

**OUTPUT:**

Conditions (Years of Speeches):

['1789-Washington.txt', '1793-Washington.txt', '1797-Adams.txt', '1801-Jefferson.txt', '1805-Jefferson.txt', '1809-Madison.txt', '1813-Madison.txt', '1817-Monroe.txt', '1821-Monroe.txt', '1825-Adams.txt']

==================================================================

Frequency of 'freedom' across different years:

1789-Washington.txt : 0

1861-Lincoln.txt : 0

1933-Roosevelt.txt : 0

2009-Obama.txt : 3

==================================================================

Most common words in '2009-Obama.txt':

[('the', 135), (',', 130), ('and', 111), ('.', 108), ('of', 82), ('to', 70), ('our', 67), ('we', 62), ('that', 49), ('a', 47)]

**B) AIM:** Find all the four-letter words in any corpus. With the help of a frequency distribution (Freq Dist), show these words in decreasing order of frequency.

**DESCRIPTION:** This program finds all the four-letter words from a selected text corpus and displays them in decreasing order of frequency using the Frequency Distribution (FreqDist) feature of the NLTK library.

A Frequency Distribution counts how many times each word appears in a given text. By filtering words of exactly four letters, we can analyze patterns of short word usage in the corpus. This type of analysis helps in understanding word popularity, text characteristics, and linguistic trends.

**PROGRAM:**

import nltk

from nltk.corpus import gutenberg

from nltk.probability import FreqDist

nltk.download('gutenberg')

words = [word.lower() for word in gutenberg.words('austen-emma.txt') if word.isalpha()]

four\_letter\_words = [word for word in words if len(word) == 4]

fdist = FreqDist(four\_letter\_words)

print("Four-letter words in decreasing order of frequency:\n")

for word, frequency in fdist.most\_common(20):

print(f"{word} : {frequency}")

**OUTPUT:**

[nltk\_data] Downloading package gutenberg to /root/nltk\_data...

[nltk\_data] Unzipping corpora/gutenberg.zip.

Four-letter words in decreasing order of frequency:

that : 1806

have : 1320

with : 1217

very : 1202

emma : 865

been : 759

were : 600

miss : 599

will : 570

must : 567

from : 546

they : 540

what : 536

this : 526

such : 489

much : 486

said : 484

more : 467

them : 432

than : 415

**C)AIM:** Define a conditional frequency distribution over the names corpus that allows you to see which initial letters are more frequent for males versus females.

**DESCRIPTION:**

This program defines a Conditional Frequency Distribution (CFD) over the Names corpus available in NLTK to analyze which initial letters (first letters of names) are more frequent for male and female names.

A Conditional Frequency Distribution shows how often certain events occur given specific conditions.  
Here,

1. The condition is the gender ('male' or 'female').
2. The event is the first letter of each name.

**PROGRAM:**

import nltk

from nltk.corpus import names

from nltk.probability import ConditionalFreqDist

nltk.download('names')

male\_names = names.words('male.txt')

female\_names = names.words('female.txt')

cfd = ConditionalFreqDist(

(gender, name[0].lower())

for gender in ['male', 'female']

for name in (male\_names if gender == 'male' else female\_names)

)

print("Most common initial letters for males:")

print(cfd['male'].most\_common(10))

print("="\*80)

print("Most common initial letters for females:")

print(cfd['female'].most\_common(10))

**OUTPUT:**

Most common initial letters for males:

[('s', 238), ('a', 213), ('m', 200), ('r', 200), ('t', 188), ('b', 173), ('c', 166), ('h', 163), ('g', 156), ('w', 151)]

==================================================================

Most common initial letters for females:

[('m', 484), ('c', 469), ('a', 443), ('l', 332), ('s', 309), ('d', 308), ('j', 293), ('k', 276), ('e', 251), ('r', 247)]

[nltk\_data] Downloading package names to /root/nltk\_data...

[nltk\_data] Unzipping corpora/names.zip.

**WEEK-14**

**A) AIM:** Write a program to identify which word has the greatest number of distinct tags? What are they, and what do they represent?

**DESCRIPTION:**

This program identifies the word that has the greatest number of distinct POS (Part-of-Speech) tags in the Brown corpus using the NLTK library.

In natural language, a single word can serve multiple grammatical roles depending on context. For example, the word “run” can be a noun (“a long run”) or a verb (“they run fast”).  
By analyzing the tagged Brown corpus, we can find which word appears with the most different POS tags — showing its linguistic versatility.

**PROGRAM:**

import nltk

from nltk.corpus import brown

nltk.download('brown')

nltk.download('universal\_tagset')

word\_tags = {}

for (word, tag) in brown.tagged\_words(tagset='universal'):

word = word.lower()

if word not in word\_tags:

word\_tags[word] = set()

word\_tags[word].add(tag)

max\_word = max(word\_tags, key=lambda w: len(word\_tags[w]))

max\_tags = word\_tags[max\_word]

print("Word with greatest number of distinct POS tags:\n")

print("Word:", max\_word)

print("Number of distinct tags:", len(max\_tags))

print("Tags:", max\_tags)

**OUTPUT:**

[nltk\_data] Downloading package brown to /root/nltk\_data...

[nltk\_data] Unzipping corpora/brown.zip.

[nltk\_data] Downloading package universal\_tagset to /root/nltk\_data...

[nltk\_data] Unzipping taggers/universal\_tagset.zip.

Word with greatest number of distinct POS tags:

Word: down

Number of distinct tags: 6

Tags: {'VERB', 'ADJ', 'PRT', 'ADP', 'ADV', 'NOUN'}

**B) AIM**: Write a program to list tags in order of decreasing frequency and what do the 20 most frequent tags represent?

**DESCRIPTION:**

This program lists all Part-of-Speech (POS) tags from the Brown corpus in decreasing order of frequency using the NLTK library.

Every word in the Brown corpus is labeled with a POS tag that identifies its grammatical role — such as noun, verb, adjective, etc. By counting how often each tag appears, we can understand which parts of speech are most common in English text.

**PROGRAM:**

import nltk

from nltk.corpus import brown

from nltk.probability import FreqDist

nltk.download('brown')

nltk.download('universal\_tagset')

tags = [tag for (word, tag) in brown.tagged\_words(tagset='universal')]

fdist = FreqDist(tags)

print("POS Tags in decreasing order of frequency:\n")

for tag, freq in fdist.most\_common(20):

print(f"{tag} : {freq}")

**OUTPUT:**

[nltk\_data] Downloading package brown to /root/nltk\_data...

[nltk\_data] Package brown is already up-to-date!

[nltk\_data] Downloading package universal\_tagset to /root/nltk\_data...

[nltk\_data] Package universal\_tagset is already up-to-date!

POS Tags in decreasing order of frequency:

NOUN : 275558

VERB : 182750

. : 147565

ADP : 144766

DET : 137019

ADJ : 83721

ADV : 56239

PRON : 49334

CONJ : 38151

PRT : 29829

NUM : 14874

X : 1386

**C) AIM:** Write a program to identify which tags are nouns most commonly found after? What do these tags represent?

**DESCRIPTION:**

This program identifies the most common POS (Part-of-Speech) tags that follow nouns in the Brown corpus using NLTK.

In English grammar, nouns are often followed by specific types of words — such as prepositions (ADP), verbs (VERB), or determiners (DET) — depending on sentence structure

**PROGRAM:**

import nltk

from nltk.corpus import brown

from nltk.probability import ConditionalFreqDist

nltk.download('brown')

nltk.download('universal\_tagset')

tags = [tag for (word, tag) in brown.tagged\_words(tagset='universal')]

bigrams = list(nltk.bigrams(tags))

cfd = ConditionalFreqDist(bigrams)

print("Most common tags that appear after NOUN:\n")

for tag, freq in cfd['NOUN'].most\_common(10):

print(f"{tag} : {freq}")

**OUTPUT:**

[nltk\_data] Downloading package brown to /root/nltk\_data...

[nltk\_data] Package brown is already up-to-date!

[nltk\_data] Downloading package universal\_tagset to /root/nltk\_data...

[nltk\_data] Package universal\_tagset is already up-to-date!

Most common tags that appear after NOUN:

. : 78326

ADP : 67460

VERB : 43819

NOUN : 41309

CONJ : 16451

ADV : 7307

PRON : 5492

PRT : 4940

DET : 4504

ADJ : 3603