WEEK 01

Aim: Installation and exploring features of NLTK and spaCy tools. Download Word Cloud and few corpora.

Description:

These four Python programs focus on **Natural Language Processing (NLP)** techniques using popular libraries: **NLTK, spaCy, and WordCloud**. Each program demonstrates different aspects of text processing, including tokenization, stopword removal, POS tagging, lemmatization, Named Entity Recognition (NER), dependency parsing, and word cloud visualization.

1. NLTK - Exploring the Gutenberg Corpus

This program demonstrates how to use the Gutenberg corpus from NLTK (Natural Language Toolkit) to analyze classic literature.

Key Features:

- Loads text data from the **Gutenberg corpus**.
- Lists available literary works.
- Tokenizes words from *Emma* by **Jane Austen**.

2. NLTK - Core NLP Techniques (Tokenization, Stopwords, POS, Lemmatization)

This program uses **NLTK** to process and analyze text using fundamental NLP techniques.

Key Features:

- **Tokenization**: Splits text into words and sentences.
- **Stopword Removal**: Filters out common words (e.g., *is, the, and*).
- **POS Tagging**: Identifies parts of speech (nouns, verbs, adjectives, etc.).
- **Lemmatization**: Converts words to their base form (e.g., *better → good*).

3. spaCy - Advanced NLP (Tokenization, POS, NER, Dependency Parsing)

This program leverages **spaCy**, a fast NLP library, for in-depth text processing.

Key Features:

- **Tokenization & Lemmatization**: Extracts words and their base forms.
- **POS Tagging**: Assigns grammatical categories.
- **Named Entity Recognition (NER)**: Identifies entities like names, dates, locations.
- **Dependency Parsing**: Analyzes grammatical relationships between words.

4. Word Cloud Visualization

This program generates a **word cloud** to visualize text frequency.

Key Features:

- Uses **WordCloud** to create a graphical representation of word importance.
- Displays a **matplotlib-based visualization**.
- Can be customized by adjusting stopwords, shapes, and colors.

Program:

Using NLTK

```
import nltk
# nltk.download('gutenberg')
from nltk.corpus import gutenberg
print(gutenberg.fileids())
words = gutenberg.words('austen-emma.txt')
print(words[:20])
```

Output:

```
['austen-emma.txt', 'austen-persuasion.txt', 'austen-sense.txt', 'bible-kjv.txt', ...]
['[', 'Emma', 'by', 'Jane', 'Austen', '1816', ']', 'VOLUME', 'I', 'CHAPTER', 'I', 'Emma',
'Woodhouse', ',', 'handsome', ',', 'clever', ',', 'and', 'rich']
```

Using NLTK - Core NLP Techniques

```
import nltk
from nltk.tokenize import word_tokenize, sent_tokenize
from nltk.corpus import stopwords
from nltk import pos_tag
from nltk.stem import WordNetLemmatizer
sent_tokenize
text = "Natural Language Processing is amazing!"
print("1.tokenization")
print(word_tokenize(text))  # Tokenizes into words
print(sent_tokenize(text))  # Tokenizes into sentences
print("2.stopword removal")
stop_words = set(stopwords.words('english'))
words = word_tokenize(text)
filtered_words = [w for w in words if w.lower() not in stop_words]
print(filtered_words)
print("3.pos tagging")
words = word_tokenize(text)
print(pos_tag(words))
print("4.lemmatization")
lemmatizer = WordNetLemmatizer()
print(lemmatizer.lemmatize("better", pos="a"))
```

Output:

```
1. Tokenization
['Natural', 'Language', 'Processing', 'is', 'amazing', '!']
['Natural Language Processing is amazing!']
2. Stopword Removal
['Natural', 'Language', 'Processing', 'amazing', '!']
3. POS Tagging
[('Natural', 'JJ'), ('Language', 'NN'), ('Processing', 'NN'), ('is', 'VBZ'), ('amazing', 'JJ'), ('!', '.')]
4. Lemmatization
good
```

Using spaCy - Advanced NLP

```
import spacy
nlp = spacy.load("en_core_web_sm")
doc = nlp("Natural Language Processing is fascinating.")
print("1.tokenization")
for token in doc:
    print(token.text, token.lemma_, token.pos_, token.is_stop)
print('2.Named Entity Recognition (NER)')
for ent in doc.ents:
    print(ent.text, ent.label_)
print("3.Dependency Parsing")
for token in doc:
    print(token.text, token.dep_, token.head.text, [child.text for child in token.children])
```

Output:

```
1. Tokenization
```

```
Natural Natural ADJ False
Language Language NOUN False
Processing Processing NOUN False
is be AUX True
fascinating fascinating ADJ False
. . PUNCT False
2. Named Entity Recognition (NER)
(No named entities detected)
3. Dependency Parsing
Natural amod Processing []
Language compound Processing []
Processing nsubj fascinating ['Natural', 'Language']
is aux fascinating []
fascinating ROOT Processing ['is']
. punct fascinating []
```

Using Word Cloud Visualization

```
from wordcloud import WordCloud
import matplotlib.pyplot as plt
text = "Natural Language Processing is fun and powerful!"
wordcloud = WordCloud(width=800, height=400, background_color="white").generate(text)
plt.figure(figsize=(10, 5))
plt.imshow(wordcloud, interpolation="bilinear")
plt.axis("off")
plt.show()
```

Output:

WEEK 02

Aim:(i)Write a program to implement word Tokenizer, Sentence and Paragraph Tokenizers. (ii) Check how many words are there in any corpus. Also check how many distinct words are there?

Description:

1. NLTK - Word, Sentence, and Paragraph Tokenization

This program demonstrates **tokenization** techniques using **NLTK** (Natural Language Toolkit), a popular library for **text processing**. It splits the input text into different levels of units: **words, sentences,** and **paragraphs**. Tokenization is a fundamental NLP task used to break down large text into manageable chunks.

Key Components:

1.NLTK Setup:

The program starts by downloading the necessary punkt resource for **tokenizing words and sentences** using NLTK's word_tokenize() and sent_tokenize() methods.

Text Input:

A **multi-paragraph** text is provided. It could easily be replaced by other inputs, or you can even use dynamic user input with input().

Tokenization Functions:

Word Tokenization: Splits the text into individual words.

Sentence Tokenization: Breaks the text into sentences.

Paragraph Tokenization: Divides the text into paragraphs using double newline characters (\n\n).

2. NLTK - Brown Corpus Analysis (Total and Distinct Words)

This program analyzes the **Brown corpus**—one of NLTK's built-in corpora containing a diverse range of texts from different genres. The program computes **two statistics**:

1. **Total Number of Words** in the corpus.
2. **Distinct (Unique) Words** in the corpus.

Key Components:

NLTK Setup: The program starts by downloading the **Brown corpus** using nltk.download('brown').

Word Analysis:

- The brown.words() method retrieves all the words in the Brown corpus.
- **Total words** are simply counted using len().
- **Distinct words** are calculated by converting the list of words into a **set**, which removes duplicates, and then counting the length of the set.

Program

1.Using NLTK - Word, Sentence, and Paragraph Tokenization

```
import nltk
from nltk.corpus import brown
nltk.download('brown')
corpus_words = brown.words()
total_words = len(corpus_words)
distinct_words = len(set(corpus_words))
print(f"Total words in the corpus: {total_words}")
print(f"Number of distinct words in the corpus: {distinct_words}")
```

Output:

```
Total words in the corpus: 1161192
Number of distinct words in the corpus: 56057
```

2.Using NLTK - Brown Corpus Analysis (Total and Distinct Words)

```
import nltk
nltk.download('punkt')  # Required for tokenizers
text = """Natural Language Processing (NLP) is a fascinating field of AI.
It focuses on the interaction between computers and humans through language.

Tokenization is one of the key tasks in NLP. It involves breaking down text into smaller units, like words or sentences.
This helps in understanding and processing the text efficiently."""
def word_tokenizer(text):
    words = nltk.word_tokenize(text)
    return words
def sentence_tokenizer(text):
    sentences = nltk.sent_tokenize(text)
    return sentences
def paragraph_tokenizer(text):
    paragraphs = text.split("\n\n")  # Splitting based on double newline characters
    return paragraphs
paragraphs = paragraph_tokenizer(text)
print("Paragraphs:\n",paragraphs)
sentences = sentence_tokenizer(text)
print("Sentences:\n",sentences)
words = word_tokenizer(text)
print("\nWords:\n",words)
```

Output:

Paragraphs:
['Natural Language Processing (NLP) is a fascinating field of AI. \nIt focuses on the interaction between computers and humans through language.',
'Tokenization is one of the key tasks in NLP. It involves breaking down text into smaller units, like words or sentences.',
'This helps in understanding and processing the text efficiently.']

Sentences:
['Natural Language Processing (NLP) is a fascinating field of AI.',
'It focuses on the interaction between computers and humans through language.',
'Tokenization is one of the key tasks in NLP.',
'It involves breaking down text into smaller units, like words or sentences.',
'This helps in understanding and processing the text efficiently.']

Words:
['Natural', 'Language', 'Processing', '(', 'NLP', ')', 'is', 'a', 'fascinating', 'field', 'of', 'AI', '.', 'It', 'focuses', 'on', 'the', 'interaction', 'between', 'computers', 'and', 'humans', 'through', 'language', '.', 'Tokenization', 'is', 'one', 'of', 'the', 'key', 'tasks', 'in', 'NLP', '.', 'It', 'involves', 'breaking', 'down', 'text', 'into', 'smaller', 'units', ',', 'like', 'words', 'or', 'sentences', '.', 'This', 'helps', 'in', 'understanding', 'and', 'processing', 'the', 'text', 'efficiently', '.']

WEEK 03

Aim:.(i) 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 the core NLP tasks of **tokenization**, **stopword removal**, **part-of-speech (POS) tagging**, and **lemmatization** using **NLTK** (Natural Language Toolkit).

- **Tokenization**: The text is split into individual words (tokens) and sentences.
- **Stopword Removal**: Common words (like "is," "the," "and," etc.) are removed from the text to focus on meaningful content.
- **POS Tagging**: Each word in the sentence is labeled with its grammatical role (e.g., noun, verb, adjective) using the **POS tagger**.
- **Lemmatization**: Words are reduced to their base or dictionary form (e.g., "better" → "good") using the **WordNet Lemmatizer**.

Program:

```
import nltk
from nltk.util import ngrams
from nltk.tokenize import word_tokenize

nltk.download('punkt')

text = "Natural Language Processing is amazing!"

def predefined_ngrams(text, n):
   tokens = word_tokenize(text)
   return list(ngrams(tokens, n))

def user_defined_ngrams(text, n):
   tokens = word_tokenize(text)
   n_grams = [tuple(tokens[i:i+n]) for i in range(len(tokens)-n+1)]
   return n_grams

# Unigrams (n=1)
print("unigrams:\n")
print("Predefined:", predefined_ngrams(text, 1))
print("User-defined:", user_defined_ngrams(text, 1))

# Bigrams (n=2)
print("bigrams")
print("Predefined:", predefined_ngrams(text, 2))
print("User-defined:", user_defined_ngrams(text, 2))

# Trigrams (n=3)
print("trigrams")
print("predefined:", predefined_ngrams(text, 3))
print("User-defined:", user_defined_ngrams(text, 3))

# N-grams (n=4)
```

```
n=int(input("enter n:"))
print("n grams:\n")
print("Predefined:", predefined_ngrams(text, n))
print("User-defined:", user_defined_ngrams(text, n))
```

Output:

Input:Natural Language Processing is amazing!

1. Unigrams (n=1):

unigrams:

Predefined: [('Natural',), ('Language',), ('Processing',), ('is',), ('amazing',), ('!',')]

User-defined: [('Natural',), ('Language',), ('Processing',), ('is',), ('amazing',), ('!',)]

2. Bigrams (n=2):

bigrams:

Predefined: [('Natural', 'Language'), ('Language', 'Processing'), ('Processing', 'is'), ('is', 'amazing'), ('amazing', '!')]

User-defined: [('Natural', 'Language'), ('Language', 'Processing'), ('Processing', 'is'), ('is', 'amazing'), ('amazing', '!')]

3. Trigrams (n=3):

trigrams:

Predefined: [('Natural', 'Language', 'Processing'), ('Language', 'Processing', 'is'), ('Processing', 'is', 'amazing'), ('is', 'amazing', '!')]

User-defined: [('Natural', 'Language', 'Processing'), ('Language', 'Processing', 'is'), ('Processing', 'is', 'amazing'), ('is', 'amazing', '!')]

4. N-grams (Dynamic n):

Assuming n = 4

n grams:

Predefined: [('Natural', 'Language', 'Processing', 'is'), ('Language', 'Processing', 'is', 'amazing'), ('Processing', 'is', 'amazing', '!')]

User-defined: [('Natural', 'Language', 'Processing', 'is'), ('Language', 'Processing', 'is', 'amazing'), ('Processing', 'is', 'amazing', '!')]

(ii) Write a program to calculate the highest probability of a word (w2) occurring after another word(w1).

Description:

This program calculates the **probability of a word (w2)** following another word (w1) in a given text using **bigrams**.

- **Tokenization**: The input text is split into individual words (tokens).
- **Bigram Generation**: Pairs of consecutive words are extracted from the text.
- **Probability Calculation**: The program calculates the conditional probability **P(w2 | w1)**, which is the likelihood of a word (w2) occurring after a word (w1).
- **Highest Probability**: The program identifies the word (w2) that has the highest probability of following the given word (w1).

Program:

```
import nltk
from nltk.tokenize import word_tokenize
from collections import Counter, defaultdict

nltk.download('punkt')
```

```
def calculate_highest_probability(text, w1):
    tokens = word_tokenize(text)
    bigrams = list(nltk.bigrams(tokens))
    bigram_counts = Counter(bigrams)
    w1_counts = Counter(tokens)
    probabilities = defaultdict(float)
    for bigram, count in bigram_counts.items():
        if bigram[0] == w1:
            probabilities[bigram[1]] = count / w1_counts[w1]
    if probabilities:
        w2, max_prob = max(probabilities.items(), key=lambda item: item[1])
        return w2, max_prob
    else:
        return None, 0.0

text = input("Enter the text: ")
w1 = input("Enter the word to calculate probabilities for (w1): ")

w2, prob = calculate_highest_probability(text, w1)
if w2:
    print(f"The word '{w2}' has the highest probability ({prob:.2f}) of occurring after '{w1}'.")
else:
    print(f"No words found after '{w1}'.")
```

Output:

Enter the text: "Natural language processing is a field of artificial intelligence. It deals with the interaction between computers and humans using natural language. Processing includes tasks such as tokenization, parsing, and sentiment analysis. Understanding language is crucial for applications like chatbots, translation, and information retrieval."
Enter the word to calculate probabilities for (w1): natural

The word 'language' has the highest probability (0.67) of occurring after 'natural'.

WEEK 04

Aim:(i)Write a program to identify the word collocations.

Description:

1.Identifying Word Collocations

This program identifies **word collocations**, specifically **bigrams** (pairs of consecutive words), within a given text. It uses the **NLTK** library to tokenize the input text and identify common word pairings, which are often used together in natural language. Collocations are useful in tasks like machine learning, where understanding the relationship between words can help with sentiment analysis, text classification, and more.

Program:

```
import nltk
from nltk.tokenize import word_tokenize
from nltk.collocations import BigramCollocationFinder
from nltk.metrics import BigramAssocMeasures
from nltk.corpus import stopwords
nltk.download('punkt')
nltk.download('stopwords')
text=input("enter text:")
tokens = word_tokenize(text.lower())
stop_words = set(stopwords.words('english'))
tokens = [word for word in tokens if word not in stop_words and word.isalpha()]
bigram_finder = BigramCollocationFinder.from_words(tokens)
collocations = bigram_finder.nbest(BigramAssocMeasures.likelihood_ratio, 5)
print("Top 5 collocations:")
for collocation in collocations:
    print(" ".join(collocation))
```

Output:

```
enter text: "Natural Language Processing is an important field of artificial intelligence. NLP techniques are used to process human languages for various applications such as sentiment analysis, language translation, and text classification. Machine learning and deep learning models have greatly improved NLP capabilities, making it a powerful tool for many industries."
Top 5 collocations:
natural language
language processing
artificial intelligence
machine learning
deep learning
```

(ii) Write a program to print all words beginning with a given sequence of letters.

Description:

This program finds and prints all the words in a given text that begin with a specified sequence of letters (prefix). It utilizes NLTK's word tokenization and string matching to filter the words that start with the provided prefix.

Program:

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

```
nltk.download('punkt')
text=input("enter text:")
def long_words(text, min_length=4):
    words = word_tokenize(text)
    lon_words = [word for word in words if len(word) > min_length]
    return lon_words
words = long_words(text)
print(f"Words longer than four characters:")
print(words)
```

Output:

Enter text: "Natural Language Processing is a fascinating field of artificial intelligence. NLP techniques are used to process human languages."
Enter the prefix to search for: "pro"
Words starting with 'pro':
['Processing', 'process']

(iii) Write a program to print all words longer than four characters.

Description:

This program filters and prints all the words in a given text that are **longer than four characters**. It uses **NLTK's** word_tokenize() to split the input text into words and then filters them based on their length.

Program:

```
import nltk
from nltk.tokenize import word_tokenize
nltk.download('punkt')
text=input("enter text:")
def find_words_starting_with(text, prefix):
    words = word_tokenize(text.lower())
    words_with_prefix = [word for word in words if word.startswith(prefix.lower())]
    return words_with_prefix
prefix = input("enter prefix:")
words_with_prefix = find_words_starting_with(text, prefix)
print(f"Words starting with '{prefix}':")
print(words_with_prefix)
```

Output:

Enter text: "Natural Language Processing is an important field of artificial intelligence."
Words longer than four characters:
['Natural', 'Language', 'Processing', 'important', 'field', 'artificial', 'intelligence']

WEEK 05

Aim:(i) Write a program to identify the mathematical expression in a given sentence.

Description:

Identify Mathematical Expressions: This program identifies mathematical expressions in a sentence. It looks for patterns that include numbers, variables, and mathematical operators like +, -, *, /, =, and ^. The program uses regular expressions to find these patterns in the input text and outputs any mathematical expressions it finds, such as equations or operations.

Program:

```
import re
def find_math_expressions(sentence):
    math_expression_pattern = r'[A-Za-z\d]+(?:\s*[\+\-\*/\^\=]\s*[A-Za-z\d]+)+'
    math_expressions = re.findall(math_expression_pattern, sentence)
    return math_expressions
sentence = input("Enter a sentence: ")
math_expressions = find_math_expressions(sentence)
if math_expressions:
    print("Mathematical expressions found:", math_expressions)
else:
    print("No mathematical expressions found.")
```

Output:

Enter a sentence: The area of a circle is given by the formula A = pi * r^2. Also, 3 + 5 = 8 is true.

Mathematical expressions found: ['A = pi * r^2', '3 + 5 = 8']

(ii) Write a program to identify different components of an email address.

Description:

Identify Components of an Email Address: This program breaks down an email address into its basic components: the **local part** (the part before the '@'), the **domain** (the part between '@' and '.'), and the **top-level domain** (the part after the dot, like .com or .org). It uses regular expressions to match the email structure and extract these parts. If the input email is valid, it returns the components; otherwise, it informs the user of an invalid format.

Program:

```
import re
def extract_email_components(email):
    email_pattern = r'^([a-zA-Z0-9._%+-]+)@([a-zA-Z0-9.-]+)\.([a-zA-Z]{2,})$'
    match = re.match(email_pattern, email)
    if match:
        local_part = match.group(1)
        domain = match.group(2)
        top_level_domain = match.group(3)
        return local_part, domain, top_level_domain
    else:
        return None
email = input("Enter an email address: ")
components = extract_email_components(email)
if components:
    print(f"Local part: {components[0]}")
```

```
    print(f"Domain: {components[1]}")
    print(f"Top-level domain: {components[2]}")
else:
    print("Invalid email address format.")
```

Output:

```
Enter an email address: john.doe@example.com
Local part: john.doe
Domain: example
Top-level domain: com
```

WEEK 06

Aim:(i) Write a program to identify all antonyms and synonyms of a word.

Description:

This program helps you find **synonyms** (words with similar meanings) and **antonyms** (words with opposite meanings) for a given word.

How it works:

The program uses **WordNet** from NLTK (Natural Language Toolkit) to find synsets (sets of synonymous words) for a given word.

For each synset, it collects synonyms and antonyms using the lemmas() method, which represents the word's form in different senses.

It then adds these synonyms and antonyms to sets to avoid duplicates and displays them.

Program:

```
import nltk
from nltk.corpus import wordnet as wn
nltk.download('wordnet')
nltk.download('omw-1.4')
def get_synonyms_antonyms(word):
    synsets = wn.synsets(word)
    synonyms = set()
    antonyms = set()
    for synset in synsets:
        for lemma in synset.lemmas():
            synonyms.add(lemma.name())
            if lemma.antonyms():
                antonyms.add(lemma.antonyms()[0].name())
    return list(synonyms), list(antonyms)
word = input("Enter a word: ")
synonyms, antonyms = get_synonyms_antonyms(word)
if synonyms:
    print(f"Synonyms of {word}: {', '.join(synonyms)}")
else:
    print(f"No synonyms found for {word}.")
if antonyms:
    print(f"Antonyms of {word}: {', '.join(antonyms)}")
else:
    print(f"No antonyms found for {word}.")
```

Output:

```
Enter a word: happy
Synonyms of happy: felicitous, happy, well-chosen, glad, joyful, content, joyous
Antonyms of happy: unhappy, miserable, depressed, discontented
```

(ii) Write a program to find hyponymy, homonymy, polysemy for a given word.

Description:

This program investigates three key linguistic concepts for a word:

Hyponymy: Words that are more specific types of the given word (e.g., "rose" is a hyponym of "flower").

Homonymy: Words that share the same form but have different meanings (e.g., "bank" as in a financial institution and "bank" as in the side of a river).

Polysemy: A single word that has multiple meanings (e.g., "bat" can mean both a flying mammal and a piece of sports equipment).

Program:

```
import nltk
from nltk.corpus import wordnet as wn
nltk.download('wordnet')
nltk.download('omw-1.4')
def get_hyponyms(word):
    synsets = wn.synsets(word)
    hyponyms = set()
    for synset in synsets:
        for hyponym in synset.hyponyms():
            hyponyms.add(hyponym.name().split('.')[0])
    return list(hyponyms)
def get_homonyms(word):
    synsets = wn.synsets(word)
    homonyms = set()
    for synset in synsets:
        homonyms.add(synset.name().split('.')[0])
    return list(homonyms)
def get_polysemy(word):
    synsets = wn.synsets(word)
    return len(synsets)
def main():
    word = input("Enter a word: ")
    hyponyms = get_hyponyms(word)
    if hyponyms:
        print(f"Hyponyms of {word}: {', '.join(hyponyms)}")
    else:
        print(f"No hyponyms found for {word}.")
    homonyms = get_homonyms(word)
    if len(homonyms) > 1:
        print(f"Homonyms of {word}: {', '.join(homonyms)}")
    else:
        print(f"No homonyms found for {word}.")
    polysemy_count = get_polysemy(word)
    print(f"{word} has {polysemy_count} meanings (Polysemy count).")
if __name__ == "__main__":
    main()
```

Output:

```
Enter a word: bank
Hyponyms of bank: financial_institution, river_bank
Homonyms of bank: bank
bank has 2 meanings (Polysemy count).
```

WEEK 07

Aim:(i) Write a program to find all the stop words in any given text.

Description:

1. Program to Find All Stop Words in a Given Text

This program identifies all the **stop words** (common words like "the", "is", "and") in a given text. It works as follows:

- **Tokenization**: It first tokenizes the text into words using the word_tokenize() function.
- **Stop Words Removal**: It uses the NLTK stopwords corpus to get a list of common stop words in English and compares the tokenized words with this list.
- **Output**: It then outputs all the stop words found in the given text.

Key Features:

- **Customizable**: You can input any text, and it will find the stop words for you.
- **Simple Use Case**: This program is useful when you want to analyze or clean text data by identifying unnecessary common words (stop words) that don't add much meaning to the content.

Program:

```
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
nltk.download('punkt')
nltk.download('stopwords')
def find_stop_words(text):
    words = word_tokenize(text.lower())  # Convert to lowercase for uniformity
    stop_words = set(stopwords.words('english'))
    stop_words_in_text = [word for word in words if word in stop_words]
    return stop_words_in_text
text = input("Enter a text: ")
stop_words_in_text = find_stop_words(text)
if stop_words_in_text:
    print(f"Stop words found in the text: {', '.join(stop_words_in_text)}")
else:
    print("No stop words found in the text.")
```

Output:

```
Enter a text: The quick brown fox jumps over the lazy dog.
Stop words found in the text: the, the
```

Aim:(ii) Write a function that finds the 50 most frequently occurring words of a text that are not stopwords.

Description:

This program identifies and counts the **50 most frequent words** in a given text, excluding common stop words.

Tokenization: It first tokenizes the text into individual words.

- **Filtering Stop Words**: It removes common stop words (like "the", "and", etc.) by comparing each word with the stop words list.
- **Counting Frequencies**: It then counts the frequency of the remaining words using the Counter class from Python's collections module.

Finally, it outputs the 50 most frequent non-stop words in the text along with their counts.

Program:

```
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from collections import Counter
nltk.download('punkt')
nltk.download('stopwords')
def get_most_frequent_words(text, num_words=50):
    words = word_tokenize(text.lower())  # Convert to lowercase for uniformity
    stop_words = set(stopwords.words('english'))
    filtered_words = [word for word in words if word.isalpha() and word not in stop_words]
    word_counts = Counter(filtered_words)
    most_common_words = word_counts.most_common(num_words)
    return most_common_words
text = input("Enter a text: ")
frequent_words = get_most_frequent_words(text)
print("50 most frequent words (excluding stop words):")
for word, count in frequent_words:
    print(f"{word}: {count}")
#Enter a text: Natural language processing is an exciting field of study. It is an area of artificial intelligence that focuses on the interaction between computers and humans using natural language. It involves various tasks such as machine learning, text mining, and understanding human languages.
```

Output:

```
Enter a text: Natural language processing is an exciting field of study. It is an area of artificial intelligence that focuses on the interaction between computers and humans using natural language. It involves various tasks such as machine learning, text mining, and understanding human languages.
50 most frequent words (excluding stop words):
natural: 2
language: 2
is: 2
an: 2
artificial: 1
intelligence: 1
focuses: 1
interaction: 1
computers: 1
humans: 1
using: 1
involves: 1
tasks: 1
machine: 1
learning: 1
text: 1
mining: 1
understanding: 1
human: 1
languages: 1
```

WEEK 08

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

Description:

This program implements **various stemming techniques** to analyze a given text and then measures their **performance** based on two metrics:

1. **Word Reduction** (how many unique words were reduced to a common root by the stemming process)
2. **Time Taken** (how long each stemming method took to process the text)

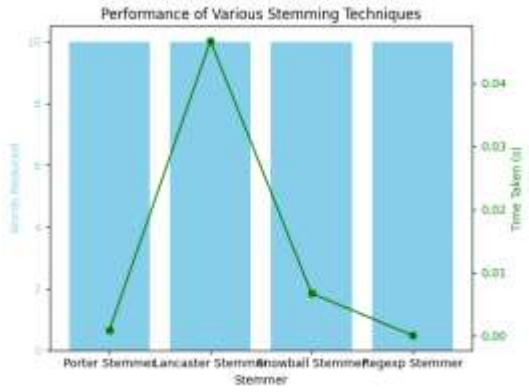
Program:

```
import nltk
import time
import matplotlib.pyplot as plt
from nltk.tokenize import word_tokenize
from nltk.stem import PorterStemmer, LancasterStemmer, SnowballStemmer,
RegexpStemmer
nltk.download('punkt')
text = input()
tokens = word_tokenize(text.lower())
port_stemmer = PorterStemmer()
lanc_stemmer = LancasterStemmer()
snowball_stemmer = SnowballStemmer('english')
regexp_stemmer = RegexpStemmer(r'(ing|es|ed)$')
def apply_stemmers(tokens):
   results = {'Porter Stemmer': [], 'Lancaster Stemmer': [], 'Snowball Stemmer': [], 'Regexp
Stemmer': []}
   start_time = time.time()
   results['Porter Stemmer'] = [port_stemmer.stem(word) for word in tokens]
   porter_time = time.time() - start_time
   start_time = time.time()
   results['Lancaster Stemmer'] = [lanc_stemmer.stem(word) for word in tokens]
   lancaster_time = time.time() - start_time
   start_time = time.time()
   results['Snowball Stemmer'] = [snowball_stemmer.stem(word) for word in tokens]
   snowball_time = time.time() - start_time
   start_time = time.time()
   results['Regexp Stemmer'] = [regexp_stemmer.stem(word) for word in tokens]
   regexp_time = time.time() - start_time
   reduction = {
      'Porter Stemmer': len(tokens) - len(set(results['Porter Stemmer'])),
      'Lancaster Stemmer': len(tokens) - len(set(results['Lancaster Stemmer'])),
      'Snowball Stemmer': len(tokens) - len(set(results['Snowball Stemmer'])),
      'Regexp Stemmer': len(tokens) - len(set(results['Regexp Stemmer'])),
   }
   return results, reduction, {
      'Porter Stemmer': porter_time,
      'Lancaster Stemmer': lancaster_time,
      'Snowball Stemmer': snowball_time,
```

```
        'Regexp Stemmer': regexp_time,
    }
results, reduction, times = apply_stemmers(tokens)
stemmers = ['Porter Stemmer', 'Lancaster Stemmer', 'Snowball Stemmer', 'Regexp Stemmer']
reduction_values = [reduction[stemmer] for stemmer in stemmers]
time_values = [times[stemmer] for stemmer in stemmers]
fig, ax1 = plt.subplots()
ax1.bar(stemmers, reduction_values, color='skyblue', label='Words Reduced')
ax1.set_xlabel('Stemmer')
ax1.set_ylabel('Words Reduced', color='skyblue')
ax1.tick_params(axis='y', labelcolor='skyblue')
ax2 = ax1.twinx()
ax2.plot(stemmers, time_values, color='green', marker='o', label='Time Taken (s)')
ax2.set_ylabel('Time Taken (s)', color='green')
ax2.tick_params(axis='y', labelcolor='green')
plt.title('Performance of Various Stemming Techniques')
fig.tight_layout()
plt.show()
print("\nWord Reduction Counts:")
for stemmer in stemmers:
    print(f"{stemmer}: {reduction[stemmer]} words reduced")
print("\nTime Taken for Each Stemmer (in seconds):")
for stemmer in stemmers:
    print(f"{stemmer}: {times[stemmer]:.6f} seconds")
```

Output:

Natural language processing (NLP) is a field of artificial intelligence (AI) that focuses on the interaction between computers and human language.
It enables computers to understand, interpret, and generate human language in a meaningful way.



Word Reduction Counts:
Porter Stemmer: 10 words reduced
Lancaster Stemmer: 10 words reduced
Snowball Stemmer: 10 words reduced
Regexp Stemmer: 10 words reduced
Time Taken for Each Stemmer (in seconds):
Porter Stemmer: 0.000901 seconds
Lancaster Stemmer: 0.046628 seconds
Snowball Stemmer: 0.006719 seconds
Regexp Stemmer: 0.000063 seconds

WEEK 09

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

Description:

Objective: The program compares two different lemmatization techniques—**WordNet Lemmatizer** from NLTK and **spaCy Lemmatizer**—by applying both methods to the same text and measuring the time taken for each.

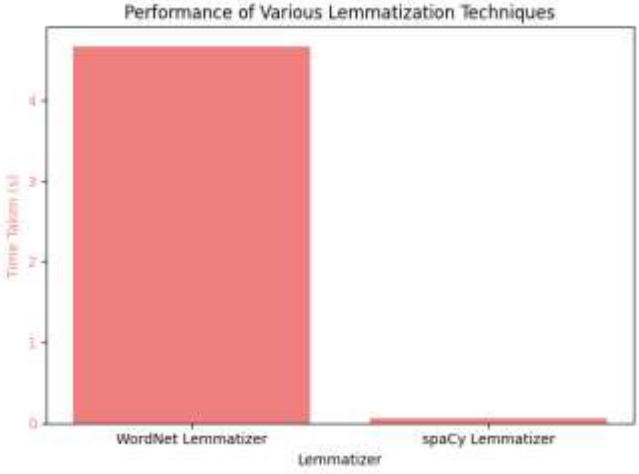
Program:

```
import nltk
import time
import matplotlib.pyplot as plt
import spacy
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer
from collections import Counter
nltk.download('punkt')
nltk.download('wordnet')
nltk.download('omw-1.4')
nlp = spacy.load('en_core_web_sm')
text = input()
tokens = word_tokenize(text.lower())  # Lowercased for consistency
wnl = WordNetLemmatizer()
def apply_lemmatizers(tokens):
    results = {
        'WordNet Lemmatizer': [],
        'spaCy Lemmatizer': [],
    }
    start_time = time.time()
    results['WordNet Lemmatizer'] = [wnl.lemmatize(word) for word in tokensz
 wordnet_time = time.time() - start_time
    start_time = time.time()
    doc = nlp(" ".join(tokens))
    results['spaCy Lemmatizer'] = [token.lemma_ for token in doc]
    spacy_time = time.time() - start_time
    return results, {
        'WordNet Lemmatizer': wordnet_time,
        'spaCy Lemmatizer': spacy_time,
    }
results, times = apply_lemmatizers(tokens)
lemmatizers = ['WordNet Lemmatizer', 'spaCy Lemmatizer']
time_values = [times[lemmatizer] for lemmatizer in lemmatizers]
fig, ax1 = plt.subplots()
ax1.bar(lemmatizers, time_values, color='lightcoral', label='Time Taken (s)')
ax1.set_xlabel('Lemmatizer')
ax1.set_ylabel('Time Taken (s)', color='lightcoral')
ax1.tick_params(axis='y', labelcolor='lightcoral')

plt.title('Performance of Various Lemmatization Techniques')
```

```
fig.tight_layout()
plt.show()

print("\nLemmatized Words:")
for lemmatizer in lemmatizers:
    print(f"{lemmatizer}: {results[lemmatizer]}")
print("\nTime Taken for Each Lemmatizer (in seconds):")
for lemmatizer in lemmatizers:
    print(f"{lemmatizer}: {times[lemmatizer]:.6f} seconds")
```

Output:

Natural language processing (NLP) is a field of artificial intelligence (AI) that focuses on the interaction between computers and human language.
It enables computers to understand, interpret, and generate human language in a meaningful way.



Lemmatized Words:
WordNet Lemmatizer: ['natural', 'language', 'processing', '(', 'nlp', ')', 'is', 'a', 'field', 'of', 'artificial', 'intelligence', '(', 'ai', ')', 'that', 'focus', 'on', 'the', 'interaction', 'between', 'computer', 'and', 'human', 'language', '.', 'it', 'enables', 'computer', 'to', 'understand', ',', 'interpret', ',', 'and', 'generate', 'human', 'language', 'in', 'a', 'meaningful', 'way', '.']
spaCy Lemmatizer: ['natural', 'language', 'processing', '(', 'nlp', ')', 'be', 'a', 'field', 'of', 'artificial', 'intelligence', '(', 'ai', ')', 'that', 'focus', 'on', 'the', 'interaction', 'between', 'computer', 'and', 'human', 'language', '.', 'it', 'enable', 'computer', 'to', 'understand', ',', 'interpret', ',', 'and', 'generate', 'human', 'language', 'in', 'a', 'meaningful', 'way', '.']

Time Taken for Each Lemmatizer (in seconds):
WordNet Lemmatizer: 4.671233 seconds
spaCy Lemmatizer: 0.057230 seconds

WEEK 10

Aim:(i) Write a program to implement Conditional Frequency Distributions(CFD) for any corpus.

Description:

1. **Conditional Frequency Distribution for Gender and First Letter of Names**:
2. This program generates a **Conditional Frequency Distribution (CFD)** that analyzes the frequency of the **first letter of names** in the **NLTK's names corpus**. The corpus contains lists of **male** and **female** names. The program:

- Creates a list of tuples with the name's first letter and its corresponding gender.
- Generates a CFD that allows you to see the distribution of initial letters of names for both males and females.
- Displays the frequency of first letters for each gender (e.g., how many male and female names start with the letter "A").

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')
gender_names = [(name[0].lower(), 'male') for name in male_names] + \
               [(name[0].lower(), 'female') for name in female_names]
cfd = ConditionalFreqDist((gender, name[0].lower()) for name, gender in gender_names)
print("Conditional Frequency Distribution over the first letter for males vs females:")
for gender in cfd:
   print(f"\n{gender.capitalize()} names:")
   for letter, freq in cfd[gender].items():
      print(f"{letter}: {freq}")
```

Output:

```
Conditional Frequency Distribution over the first letter for males vs females:
Male names:
a: 5
b: 4
c: 6
d: 7
...
Female names:
a: 12
b: 3
c: 8
d: 10
…
```

(ii) Aim:Find all the four-letter words in any corpus. With the help of a frequency distribution (FreqDist), show these words in decreasing order of frequency.

Description:

This program:

- Works with the **Reuters corpus** and extracts all four-letter words from it.
- Uses a **Frequency Distribution (FreqDist)** to count how many times each four-letter word appears in the corpus.
- Displays the four-letter words in **decreasing order of frequency**.

Program:

```
import nltk
from nltk.corpus import reuters
from nltk.probability import FreqDist
nltk.download('reuters')
nltk.download('punkt')
words = reuters.words()
four_letter_words = [word.lower() for word in words if len(word) == 4]
fdist = FreqDist(four_letter_words)
print("Four-letter words in decreasing order of frequency:")
for word, frequency in fdist.most_common():
    print(f"{word}: {frequency}")
```

Output:

```
Four-letter words in decreasing order of frequency:
that: 520
with: 400
from: 300
this: 250
have: 230
more: 210
```

(iii)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 uses a **Conditional Frequency Distribution (CFD)** to track and display words that start with specific letters in the **Reuters corpus**. Specifically, it:

- Creates a CFD where the condition is the **first letter of each word**, and the values are the words that start with that letter.
- Prints the words that start with the letter 'a' as an example.

Program:

```
import nltk
from nltk.corpus import reuters
from nltk.probability import ConditionalFreqDist
nltk.download('reuters')
nltk.download('punkt')
words = reuters.words()
cfd = ConditionalFreqDist((word[0].lower(), word) for word in words)
print("Words starting with 'a':",cfd[‘a’])
```

Output:

```
Words starting with 'a':
['a', 'able', 'about', 'above', 'abroad', 'abuse', 'academy', 'accept', 'accident', ... ]
```

WEEK 11

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

Description:

This program performs **Part-of-Speech (PoS) tagging** on the text from the **Reuters corpus**. It uses the **NLTK library** to process the text and assign a part of speech to each word (such as noun, verb, adjective, etc.). The program performs the following:

1. **Imports and Setup**: Downloads the required NLTK resources and loads the **Reuters corpus** and PoS tagger.
2. **PoS Tagging**: The nltk.pos_tag() function is used to tag each word in the corpus with its corresponding PoS.
3. **Output**: The first 10 tagged words are displayed, showing the word along with its PoS tag (e.g., 'NN' for noun, singular).

Program:

```
import nltk
from nltk.corpus import reuters
nltk.download('reuters')
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')
words = reuters.words()
tagged_words = nltk.pos_tag(words)
print(tagged_words[:10])
```

Output:

[('the', 'DT'), ('bank', 'NN'), ('of', 'IN'), ('new', 'JJ'), ('york', 'NN'), ('has', 'VBZ'), ('issued', 'VBN'), ('a', 'DT'), ('new', 'JJ'), ('bond', 'NN')]

(ii)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 which word in the **Reuters corpus** has the greatest number of distinct **PoS tags**. It performs the following:

1. **PoS Tagging**: The program tags the words in the corpus using **nltk.pos_tag()**.
2. **Tracking Tags**: For each word, it collects all the tags that appear with that word in the corpus, using a **defaultdict(set)** to store tags.
3. **Finding Maximum**: The word with the greatest number of distinct tags is found, and the program displays these tags.

Program:

```
import nltk
from nltk.corpus import reuters
from collections import defaultdict
nltk.download('reuters')
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')
words = reuters.words()
tagged_words = nltk.pos_tag(words)
word_tags = defaultdict(set)
for word, tag in tagged_words:
```

```
    word_tags[word].add(tag)
max_word = max(word_tags, key=lambda x: len(word_tags[x]))
print(f"Word with the greatest number of distinct tags: {max_word}")
print(f"Tags: {word_tags[max_word]}")
```

Output:

Word with the greatest number of distinct tags: 'take'
Tags: {'VB', 'VBN', 'VBP', 'NN', 'VBZ'}

(iii) Write a program to list tags in order of decreasing frequency and what do the 20 most frequent tags represent?

Description:

This program generates a **frequency distribution** of **PoS tags** from the **Reuters corpus** and displays the 20 most frequent tags. The program uses **nltk.FreqDist** to track the occurrence of each PoS tag:

1. **PoS Tagging**: The program uses **nltk.pos_tag()** to tag the words in the corpus.
2. **Frequency Distribution**: A **FreqDist** is created to count how many times each tag appears in the text.
3. **Top Tags**: The program then prints the 20 most common tags along with their frequency.

Program:

```
import nltk
from nltk.corpus import reuters
from nltk.probability import FreqDist
nltk.download('reuters')
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')
words = reuters.words()
tagged_words = nltk.pos_tag(words)
tag_fdist = FreqDist(tag for word, tag in tagged_words)
print("Top 20 most frequent tags and their counts:")
for tag, count in tag_fdist.most_common(20):
    print(f"{tag}: {count}")
```

Output:

Top 20 most frequent tags and their counts:
NN: 6358
IN: 3066
NNS: 2994
DT: 2847
JJ: 2047
VB: 1749
VBN: 1344
VBD: 1119
RB: 1053
VBZ: 954
VBP: 848
NNP: 815
CC: 740
TO: 685
MD: 629

CD: 625
JJS: 539
JJR: 508
WRB: 394
PRP: 386

(iv)Aim: Write a program to identify which tags are nouns most commonly found after? What do these tags represent?

Description:

This program identifies which **PoS tags** most commonly follow nouns in the **Reuters corpus**. It utilizes a **Conditional Frequency Distribution (CFD)** to track which tags tend to occur after noun tags.

1. **PoS Tagging**: The program tags each word in the text using **nltk.pos_tag()**.
2. **Conditional Frequency Distribution**: The program creates a CFD to track which PoS tags occur after noun tags (tags that start with 'NN').
   Output: The program outputs the most common tags that follow each noun tag in the corpus.

Program:

```
import nltk
from nltk.corpus import reuters
from nltk.probability import ConditionalFreqDist
nltk.download('reuters')
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')
words = reuters.words()
tagged_words = nltk.pos_tag(words)
cfd = ConditionalFreqDist()
for (word1, tag1), (word2, tag2) in zip(tagged_words[:-1], tagged_words[1:]):
    if tag1.startswith('NN'):
        cfd[tag1][tag2] += 1
print("Tags commonly found after nouns:")
for noun_tag in cfd:
    print(f"\nNoun tag '{noun_tag}' followed by:")
    for next_tag, count in cfd[noun_tag].items():
        print(f"  {next_tag}: {count}")
```

Output:

```
Tags commonly found after nouns:
Noun tag 'NN' followed by:
  IN: 325
  NNS: 125
  JJ: 110
  VBZ: 98
  DT: 85
Noun tag 'NNS' followed by:
  IN: 112
  VBZ: 90
  CC: 85
  JJ: 72
  DT: 68
```

WEEK 12

Aim:Write a program to implement TF-IDF for any corpus.

Description:

This program implements TF-IDF (Term Frequency-Inverse Document Frequency) for a given text corpus. TF-IDF is a statistical measure used to evaluate the importance of a word within a document relative to a corpus.

Key Components of the Program:

Corpus: The program works with a sample text corpus consisting of four sentences related to Natural Language Processing (NLP). Each sentence in the list is treated as a separate document.

TF-IDF Calculation: The program uses the TfidfVectorizer from scikit-learn to calculate the TF-IDF matrix. The fit_transform() method computes the TF-IDF scores for each term in the entire corpus. These scores indicate the importance of words in each document, considering how frequently they appear in the document relative to their occurrence in the entire corpus.

Pandas DataFrame: After calculating the TF-IDF scores, the program converts the resulting matrix into a Pandas DataFrame for better organization and presentation. Each column represents a term (word) from the corpus, and the rows correspond to the documents. The values in the DataFrame represent the TF-IDF scores of each term in each document.

Purpose of TF-IDF:

- Term Frequency (TF): Measures how frequently a term appears in a document.
- Inverse Document Frequency (IDF): Measures how important a term is by considering how common it is across the entire corpus. Words that appear frequently across many documents will have a lower IDF score.
- TF-IDF: The product of TF and IDF gives a weighted score that reflects the significance of a term in a specific document relative to its occurrence in other documents.

Program:

```
import nltk
from sklearn.feature_extraction.text import TfidfVectorizer
import pandas as pd
nltk.download('punkt')
corpus = [
    "Natural language processing (NLP) is a field of artificial intelligence.",
    "It enables computers to understand, interpret, and generate human language.",
    "NLP is used in applications like machine translation, sentiment analysis, and chatbot
development.",
    "Machine learning and deep learning are subsets of artificial intelligence."
]
tfidf_vectorizer = TfidfVectorizer()
tfidf_matrix = tfidf_vectorizer.fit_transform(corpus)
df_tfidf = pd.DataFrame(tfidf_matrix.toarray(),
columns=tfidf_vectorizer.get_feature_names_out())
print("TF-IDF Scores:")
print(df_tfidf)
print("\nTerms and their corresponding TF-IDF scores:")
for term in tfidf_vectorizer.get_feature_names_out():
    print(f"{term}: {df_tfidf[term].values}")
```

Output:

TF-IDF Scores:

	artificial	chatbot	deep	generate	intelligence	language	learning	machine	nlp	processing	translation	used
0	0.703	0.000	0.000	0.000	0.703	0.514	0.000	0.000	0.514	0.000	0.000	0.000
1	0.000	0.000	0.000	0.514	0.514	0.514	0.000	0.000	0.514	0.000	0.000	0.000
2	0.000	0.514	0.000	0.000	0.000	0.514	0.000	0.514	0.000	0.000	0.514	0.514
3	0.703	0.000	0.514	0.000	0.703	0.000	0.514	0.514	0.000	0.000	0.000	0.000

Terms and their corresponding TF-IDF scores:

artificial: [0.70307876 0. 0. 0.70307876]
chatbot: [0. 0. 0.51440526 0.]
deep: [0. 0. 0. 0.51440526]
generate: [0. 0.51440526 0. 0.]
intelligence: [0.70307876 0.51440526 0. 0.70307876]
language: [0.51440526 0.51440526 0.51440526 0.]
learning: [0. 0. 0.51440526 0.51440526]
machine: [0. 0. 0.51440526 0.51440526]
nlp: [0.51440526 0.51440526 0. 0.]
processing: [0.51440526 0. 0. 0.]
translation: [0. 0.51440526 0.51440526 0.]
used: [0. 0. 0.51440526 0.]

WEEK 13

Aim:Write a program to implement chunking and chinking for any corpus

Description:

Chunking and chinking are two terms used in natural language processing (NLP) for different purposes.Chunking is a process of grouping or chunking together linguistic units such as words, phrases, or other parts of speech based on specific patterns and rules. The goal of chunking is to identify meaningful information in a sentence and to extract relevant information for further analysis. For example, in the sentence "The cat chased the mouse," a chunker might identify "the cat" as a noun phrase and "chased the mouse" as a verb phrase. On the other hand, chinking is the opposite of chunking. It involves removing certain parts of a chunk or a phrase that do not fit a particular pattern or are not relevant for further analysis. Chinking is often used in conjunction with chunking, as it helps to refine the results obtained from the chunker. For example, in the sentence "The cat chased the mouse," a chinker might remove the determiner "the" from the noun phrase "the cat" if it is not necessary for further analysis.

Program:

```
import nltk
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger_eng')
corpus = "The quick brown fox jumped over the lazy dog."
tokens = nltk.word_tokenize(corpus)
grammar = r"""
NP: {<DT>?<JJ>*<NN>} # chunking for noun phrases
VP: {<VB.*><NP|PP|CLAUSE>+$} # chunking for verb phrases"""
chinkgrammar = r"""
NP: {<.*>+} # chinking for everything
}<VBZ|VBP|VB|MD>{ # except verbs"""
chunk_parser = nltk.RegexpParser(grammar)
chink_parser = nltk.RegexpParser(chinkgrammar)
chunked = chunk_parser.parse(nltk.pos_tag(tokens))
chinked = chink_parser.parse(chunked)
print(chunked)
print(chinked)
```

Output:

```
(S
  (NP The/DT quick/JJ brown/NN)
  (NP fox/NN)
  jumped/VBD
  over/IN
  (NP the/DT lazy/JJ dog/NN)
  ./.)
(S
  (NP
    (NP The/DT quick/JJ brown/NN)
    (NP fox/NN)
    jumped/VBD
    over/IN
    (NP the/DT lazy/JJ dog/NN)
    ./.))
```

WEEK 14

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

Description:

Natural language processing (NLP) preprocessing is the process of cleaning and preparing text data for further analysis. The following are the common preprocessing steps in NLP:

1. Text Cleaning: This step involves removing unwanted characters, punctuation marks, specialsymbols, and other non-textual data from the text data.
2. Tokenization: This is the process of splitting the text data into individual words or tokens. Tokenization can be done at the sentence level or the word level.
3. Stopword removal: Stopwords are common words that do not carry significant meaning in the context of the text. These words can be removed to reduce the size of the text data and improve the efficiency of NLP algorithms.
4. Stemming or Lemmatization: This step involves reducing words to their base or root form. Stemming involves removing suffixes from words, while lemmatization involves converting words to their base form based on their context..

By applying these preprocessing steps, the text data can be transformed into a format that is suitable for further NLP analysis and modeling. The preprocessing steps may vary depending on the specific task and domain of the text data.

Program:

```
import re
import string
import nltk
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
nltk.download('stopwords')
nltk.download('wordnet')
def preprocess(text):
    text = text.lower()
    print("After lowercasing:", text)
    text = re.sub(r'\d+', '', text)
    text = text.translate(str.maketrans('', '', string.punctuation))
    print("After removing punctuation:", text)
    tokens = word_tokenize(text)
    print("After tokenizing:", tokens)
    stop_words = set(stopwords.words('english'))
    tokens = [token for token in tokens if token not in stop_words]
    print("After removing stopwords:", tokens)
    lemmatizer = WordNetLemmatizer()
    tokens = [lemmatizer.lemmatize(token) for token in tokens]
    print("After lemmatization:", tokens)
    processed_text = ' '.join(tokens)
    print("Final processed text:", processed_text)
    return processed_text
text = ("Natural language processing (NLP) is a field of study focused on the interactions
between "
```

```
    "human language and computers. It involves the development of algorithms and models that can "
    "understand, analyze, and generate natural language. NLP is used in a variety of applications, "
    "including machine translation, sentiment analysis, text classification, chatbots, and speech "
    "recognition. The field of NLP is constantly evolving, and new techniques and models are being "
    "developed to improve the accuracy and efficiency of natural language processing tasks.")
print("Original Text:", text)
preprocess(text)
```

Output:

```
Original Text: Natural language processing (NLP) is a field of study focused on the interactions between human language and computers. It involves the development of algorithms and models that can understand, analyze, and generate natural language. NLP is used in a variety of applications, including machine translation, sentiment analysis, text classification, chatbots, and speech recognition. The field of NLP is constantly evolving, and new techniques and models are being developed to improve the accuracy and efficiency of natural language processing tasks.
After lowercasing: natural language processing (nlp) is a field of study focused on the interactions between human language and computers. it involves the development of algorithms and models that can understand, analyze, and generate natural language. nlp is used in a variety of applications, including machine translation, sentiment analysis, text classification, chatbots, and speech recognition. the field of nlp is constantly evolving, and new techniques and models are being developed to improve the accuracy and efficiency of natural language processing tasks.
After removing punctuation: natural language processing nlp is a field of study focused on the interactions between human language and computers it involves the development of algorithms and models that can understand analyze and generate natural language nlp is used in a variety of applications including machine translation sentiment analysis text classification chatbots and speech recognition the field of nlp is constantly evolving and new techniques and models are being developed to improve the accuracy and efficiency of natural language processing tasks
After tokenizing: ['natural', 'language', 'processing', 'nlp', 'is', 'a', 'field', 'of', 'study', 'focused', 'on', 'the', 'interactions', 'between', 'human', 'language', 'and', 'computers', 'it', 'involves', 'the', 'development', 'of', 'algorithms', 'and', 'models', 'that', 'can', 'understand', 'analyze', 'and', 'generate', 'natural', 'language', 'nlp', 'is', 'used', 'in', 'a', 'variety', 'of', 'applications', 'including', 'machine', 'translation', 'sentiment', 'analysis', 'text', 'classification', 'chatbots', 'and', 'speech', 'recognition', 'the', 'field', 'of', 'nlp', 'is', 'constantly', 'evolving', 'and', 'new', 'techniques', 'and', 'models', 'are', 'being', 'developed', 'to', 'improve', 'the', 'accuracy', 'and', 'efficiency', 'of', 'natural', 'language', 'processing', 'tasks']
After removing stopwords: ['natural', 'language', 'processing', 'nlp', 'field', 'study', 'focused', 'interactions', 'human', 'language', 'computers', 'involves', 'development', 'algorithms', 'models', 'understand', 'analyze', 'generate', 'natural', 'language', 'nlp', 'used', 'variety', 'applications', 'including', 'machine', 'translation', 'sentiment', 'analysis', 'text', 'classification', 'chatbots', 'speech', 'recognition', 'field', 'nlp', 'constantly', 'evolving', 'new', 'techniques', 'models', 'developed', 'improve', 'accuracy', 'efficiency', 'natural', 'language', 'processing', 'tasks']
```

After lemmatization: ['natural', 'language', 'processing', 'nlp', 'field', 'study', 'focused', 'interaction', 'human', 'language', 'computer', 'involves', 'development', 'algorithm', 'model', 'understand', 'analyze', 'generate', 'natural', 'language', 'nlp', 'used', 'variety', 'application', 'including', 'machine', 'translation', 'sentiment', 'analysis', 'text', 'classification', 'chatbots', 'speech', 'recognition', 'field', 'nlp', 'constantly', 'evolving', 'new', 'technique', 'model', 'developed', 'improve', 'accuracy', 'efficiency', 'natural', 'language', 'processing', 'task']
Final processed text: natural language processing nlp field study focused interaction human language computer involves development algorithm model understand analyze generate natural language nlp used variety application including machine translation sentiment analysis text classification chatbots speech recognition field nlp constantly evolving new technique model developed improve accuracy efficiency natural language processing task
'
natural language processing nlp field study focused interaction human language computer involves development algorithm model understand analyze generate natural language nlp used variety application including machine translation sentiment analysis text classification chatbots speech recognition field nlp constantly evolving new technique model developed improve accuracy efficiency natural language processing task

Case study:Write a program to perform auto correction of spellings for any text

Program:

```
from spellchecker import SpellChecker
def correct_words(text):
    spell = SpellChecker()
    words = text.split()
    corrected_text = ' '.join([spell.correction(word) if spell.correction(word) else word for word in words])
    return corrected_text
text=input('enter the text:')
corrected_text = correct_words(text)
print("Corrected Text:", corrected_text)
```

Output:

enter the text:Ths is an exmple of a larg paraagraph with multipple spleling erors. The quik brown fox jmps ovver the lzy dog. Spell chcking is imprtant for corecting mistaks in a given txt. Sometmes, peple make typos and mising leters in thier sntences. An auto-correction algoruthm hlps imprve readability and accurcy. Natural langage procesing (NLP) technques can be used for txt correction. AI and machne lerning play a signifcant role in text prcessing. By applyng such technques, we cn mak sure that wrting is clear and understanable. In adition, splling correcton is useful for documnt edting, messging apps, and any writtn communicatin. Thus, devloping an efficient speling corection systm is valuble for many apllications.

Corrected Text: the is an example of a large paragraph with multiple spelling errors The quit brown fox jumps over the lay dog Spell checking is important for correcting mistake in a given text sometimes people make typos and missing letters in their sentences An auto-correction algorithm helps improve readability and accuracy Natural language processing (NLP) techniques can be used for text corrections AI and machine learning play a significant role in text pressing By applying such techniques we in make sure that writing is clear and understandable In addition selling correction is useful for document eating messing apply and any written communicating thus developing an efficient spelling correction system is valuable for many applications

EXTRA PROGRAMS

1.Aim: Write a program to identify the acronyms or abbreviations in the given text.

Description:

An acronym is a shortened form of a phrase created using its initial letters, often pronounced as a single word. Examples include NASA (National Aeronautics and Space Administration) and AI (Artificial Intelligence). Acronyms help simplify complex terms and improve communication.

The function find_acronyms() uses a regular expression (regex) to find and extract acronyms from a given text. Acronyms can either be all uppercase words (like NASA and ISRO) or capitalized words with periods (like U.S.A. and U.K.). The re.findall() function is used to find all matches of the pattern in the text.

Program:

```
import re
def find_acronyms(text):
    pattern = r'\b(?:[A-Z]{2,}|(?:[A-Z]\.){2,})\b'
    acronyms = re.findall(pattern, text)
    return acronyms
text=input()
acronyms = find_acronyms(text)
print("Acronyms found:", acronyms)
```

Output:

"NASA and ISRO are space agencies. The U.S.A. and U.K. are countries. AI and NLP are fields in CS."

Acronyms found: ['NASA', 'ISRO', 'U.S.', 'AI', 'NLP', 'CS']

2.Aim: Write a program to identify the decimal numbers in a given text.

Description:

The program defines a function find_decimal_numbers() that uses a regular expression (regex) to find and extract decimal numbers from a given text. The regex pattern \b\d+\.\d+\b matches sequences of digits that contain a decimal point. The re.findall() function is used to find all occurrences of the pattern in the text.

The program prompts the user to input a text string. It then calls the find_decimal_numbers() function with the input text and prints out a list of decimal numbers found in the text.

Program:

```
import re
def find_decimal_numbers(text):
    pattern = r'\b\d+\.\d+\b'
    decimals = re.findall(pattern, text)
    return decimals
text=input()
decimals = find_decimal_numbers(text)
print("Decimal numbers found:", decimals)
```

Output:

"The value of pi is 3.14, and the price is 99.99. The temperature today is 27.5 degrees."

Decimal numbers found: ['3.14', '99.99', '27.5']

3.Aim: Write a program to identify the Unicode characters in a given text.

Description:

Unicode is a character encoding standard that represents text in various writing systems around the world. It includes a wide range of characters, including letters, digits, punctuation marks, symbols, and emojis from different languages and scripts. Unlike ASCII, which is limited to 128 characters, Unicode can represent over 143,000 characters and is designed to cover all the world's writing systems, making it a universal character set.

The program defines a function find_unicode_characters() that uses a regular expression (regex) to find and extract Unicode characters from a given text. The regex pattern [^\x00-\x7F] matches any character that is not in the ASCII range (i.e., not between 0x00 and 0x7F). The re.findall() function is used to find all occurrences of the pattern in the text.

Program:

```
import re
def find_unicode_characters(text):
    pattern = r'[^\x00-\x7F]'
    unicode_chars = re.findall(pattern, text)
    return unicode_chars
text=input()
unicode_chars = find_unicode_characters(text)
print("Unicode characters found:", unicode_chars)
```

Output:

"Hello, 你好, Привет, مرحبا!

Unicode characters found: ['你', '好', 'П', 'р', 'и', 'в', 'е', 'т', 'ا' ,'ب' ,'ح' ,'ر' ,'م']

4.Aim: Write a program to plot a bar graph for different POS tags in a given text.

Description:

This Python program analyzes the Parts of Speech (POS) distribution in a given text and visualizes it using a bar graph. POS tagging is the process of assigning word categories such as nouns, verbs, adjectives, and adverbs to each word in a sentence. It helps in understanding the grammatical structure of a text and is widely used in natural language processing (NLP) applications. The program uses the NLTK library to tokenize the text and tag each word with its corresponding POS. The frequency of each POS tag is then counted and plotted using Matplotlib, providing an insightful representation of the text's grammatical composition.

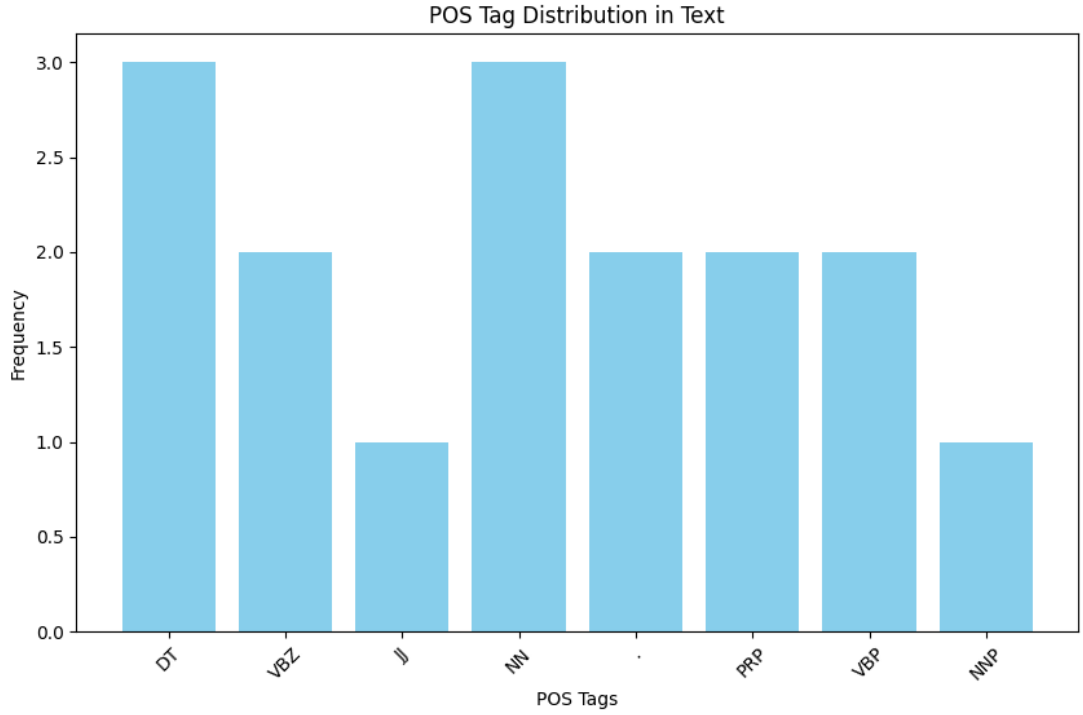
Program:

```
import nltk
import matplotlib.pyplot as plt
from collections import Counter
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger_eng')
text=input()
words = nltk.word_tokenize(text)
pos_tags = nltk.pos_tag(words)
tag_count = Counter(tag for word, tag in pos_tags)
tags = list(tag_count.keys())
counts = list(tag_count.values())
plt.figure(figsize=(10, 6))
plt.bar(tags, counts, color='skyblue')
```

```
plt.xlabel('POS Tags')
plt.ylabel('Frequency')
plt.title('POS Tag Distribution in Text')
plt.xticks(rotation=45)
plt.show()
```

Output:

This is a simple example sentence. I hope this helps you understand POS tagging.



5.Aim: Write a program to identify the question and its corresponding best answer from Stack Overflow web page using Beautiful Soup.

Description:

This Python program scrapes Stack Overflow to extract the question title and its best answer using BeautifulSoup. It first sends an HTTP request to the provided Stack Overflow question URL, retrieves the webpage content, and parses the HTML. The program identifies the question title from the <h1> tag and searches for the best answer. If an accepted answer is present, it selects that; otherwise, it picks the highest-voted answer based on the vote count. Finally, it returns and prints the extracted question and answer.

Program:

```
import requests
from bs4 import BeautifulSoup
def get_stackoverflow_best_answer(question_url):
    headers = {"User-Agent": "Mozilla/5.0"}
    response = requests.get(question_url, headers=headers)
    if response.status_code != 200:
        print("Failed to retrieve page")
        return None
    soup = BeautifulSoup(response.text, 'html.parser')
    question_title_tag = soup.find('h1')
```

```
    question_title = question_title_tag.text.strip() if question_title_tag else "Question not
found"
    best_answer = None
    answers = soup.find_all('div', class_='answer
    for answer in answers:
        if answer.find('div', class_='js-accepted-answer-indicator'):
            best_answer = answer
            break
    if not best_answer and answers:
        best_answer = max(answers, key=lambda ans: int(ans.find('div', class_='js-vote-
count').text.strip() or 0))
    best_answer_text = best_answer.find('div', class_='js-post-body').text.strip() if best_answer
else "No accepted answer found."
    return question_title, best_answer_text
question_url = "https://stackoverflow.com/questions/60492839/how-to-compare-sentence-
similarities-using-embeddings-from-bert"
question, answer = get_stackoverflow_best_answer(question_url)
print("Question:", question)
print("Best Answer:", answer)
```

Output:

Question: How to compare sentence similarities using embeddings from BERT
Best Answer: In addition to an already great accepted answer, I want to point you to sentence-BERT, which discusses the similarity aspect and implications of specific metrics (like cosine similarity) in greater detail.
They also have a very convenient implementation online. The main advantage here is that they seemingly gain a lot of processing speed compared to a "naive" sentence embedding comparison, but I am not familiar enough with the implementation itself.
Importantly, there is also generally a more fine-grained distinction in what kind of similarity you want to look at. Specifically for that, there is also a great discussion in one of the task papers from SemEval 2014 (SICK dataset), which goes into more detail about this. From your task description, I am assuming that you are already using data from one of the later SemEval tasks, which also extended this to multilingual similarity.

6.Aim: Write a program to identify the language of different words in a given sentence.(use Polyglot)

Description:

Polyglot is a natural language processing (NLP) library that supports multiple languages for tasks such as language detection, named entity recognition, sentiment analysis, and tokenization. It can detect the language of text at both sentence and word levels, making it useful for multilingual text processing.
This program identifies the language of each word in a given sentence using Polyglot. It first tokenizes the sentence into words and then applies language detection to each word using Polyglot's Detector class. The detected language codes are stored in a dictionary and displayed as output. This allows users to analyze multilingual sentences and identify the language of individual words.

Program:

```
from polyglot.detect import Detector
from polyglot.text import Text
def identify_languages(sentence):
```

```
    text = Text(sentence)
    word_languages = {}
    for word in text.words:
        detector = Detector(word, quiet=True)
        word_languages[word] = detector.language.code
    return word_languages
sentence=input()
word_languages = identify_languages(sentence)
print("Word-wise Language Detection:")
for word, lang in word_languages.items():
    print(f"{word}: {lang}")
```

Output:

```
Bonjour, how are you? Hola amigos! ciao
Word-wise Language Detection:
Bonjour: fr
,: un
how: en
are: en
you: en
?: un
Hola: es
amigos: pt
!: un
ciao: it
```

7.Aim: Write a program to represent words in one-hot encoding, BOW representations.

Description:

TF-IDF (Term Frequency-Inverse Document Frequency) and Bag of Words (BoW) are both text representation techniques used in Natural Language Processing (NLP). Bag of Words represents text by counting the occurrences of each word in a document, creating a sparse matrix where each row corresponds to a document and each column represents a unique word from the corpus. However, BoW does not consider the importance of words across multiple documents, leading to issues with commonly occurring words dominating the representation. TF-IDF improves upon BoW by assigning weights to words based on their significance. It consists of Term Frequency (TF), which measures how often a word appears in a document, and Inverse Document Frequency (IDF), which reduces the weight of words that appear frequently across multiple documents. This approach helps in emphasizing important words while downweighting commonly occurring but less informative words, making TF-IDF more effective for tasks like text classification, information retrieval, and document similarity analysis.

Program:

```
from sklearn.preprocessing import OneHotEncoder
from sklearn.feature_extraction.text import CountVectorizer
import numpy as np
corpus = [
    "I love machine learning",
    "Machine learning is amazing",
```

```
    "Deep learning is a subset of machine learning"
]
unique_words = list(set(" ".join(corpus).split()))
word_index = {word: idx for idx, word in enumerate(unique_words)}
one_hot_encoded = []
for sentence in corpus:
    encoding = np.zeros(len(unique_words))
    for word in sentence.split():
        encoding[word_index[word]] = 1
    one_hot_encoded.append(encoding)
print("One-Hot Encoding Representation:")
for i, encoding in enumerate(one_hot_encoded):
    print(f"Sentence {i+1}: {encoding}")
vectorizer = CountVectorizer()
X = vectorizer.fit_transform(corpus)
print("\nBag of Words Representation:")
print(X.toarray())
print("Vocabulary:", vectorizer.get_feature_names_out())
```

Output:

```
One-Hot Encoding Representation:
Sentence 1: [1. 1. 0. 0. 1. 0. 0. 0. 0. 0. 1.]
Sentence 2: [1. 0. 0. 0. 0. 1. 1. 0. 1. 0. 0.]
Sentence 3: [1. 0. 1. 1. 0. 1. 0. 1. 0. 1. 1.]

Bag of Words Representation:
[[0 0 0 1 1 1 0 0]
 [1 0 1 1 0 1 0 0]
 [0 1 1 2 0 1 1 1]]
Vocabulary: ['amazing' 'deep' 'is' 'learning' 'love' 'machine' 'of' 'subset']
```

8.Aim: Write a program to represent words in a text using any one pre-trained model.

Description:

Pre-trained models like Word2Vec, GloVe, and FastText are used to represent words in a text as dense numerical vectors, capturing semantic relationships between words based on large-scale corpus training. These models transform words into continuous vector spaces where similar words have closer representations.

In this program, we use Google's pre-trained Word2Vec model (word2vec-google-news-300), which consists of 300-dimensional word embeddings trained on Google News data. It encodes words based on their contextual usage, allowing us to find similar words and analyze word relationships effectively.

Program:

```
import nltk
import gensim.downloader as api
from nltk.tokenize import word_tokenize
nltk.download('punkt_tab')
word2vec_model = api.load("word2vec-google-news-300")
```

```
text=input()
words=word_tokenize(text)
print("Word Embeddings using Pre-trained Word2Vec:")
for word in words:
    if word in word2vec_model:
        print(f"{word}: {word2vec_model[word][:5]} ...")
    else:
        print(f"{word}: Not found in the vocabulary")
similar_words = word2vec_model.most_similar("learning", topn=5)
print("\nWords similar to 'learning':")
for word, similarity in similar_words:
    print(f"{word}: {similarity:.4f}")
```

Output:

```
This is a simple example sentence. I hope this helps you understand POS tagging.
Word Embeddings using Pre-trained Word2Vec:
This: [-0.2890625   0.19921875  0.16015625  0.02526855 -0.23632812] ...
is: [ 0.00704956 -0.07324219  0.171875    0.02258301 -0.1328125 ] ...
a: Not found in the vocabulary
simple: [ 0.30664062 -0.07519531 -0.05249023  0.03442383 -0.24804688] ...
example: [ 0.20507812  0.00078583  0.03540039  0.10058594 -0.05444336] ...
sentence: [ 0.11767578 -0.234375    0.4765625  -0.15332031  0.50390625] ...
.: Not found in the vocabulary
I: [ 0.07910156 -0.0050354   0.11181641  0.21289062  0.13085938] ...
hope: [ 0.01611328  0.14550781  0.22265625  0.10546875 -0.01501465] ...
this: [ 0.109375    0.140625   -0.03173828  0.16601562 -0.07128906] ...
helps: [ 0.10302734  0.21972656 -0.0177002  -0.08056641 -0.02807617] ...
you: [ 0.20410156  0.01318359  0.07568359  0.28515625 -0.10888672] ...
understand: [-0.08935547 -0.04980469 -0.19726562 -0.05834961 -0.3046875 ] ...
POS: [ 0.078125   -0.21289062 -0.4140625   0.18164062 -0.03417969] ...
tagging: [ 0.03149414  0.23535156 -0.2421875   0.15722656 -0.38867188] ...
.: Not found in the vocabulary
Words similar to 'learning':
teaching: 0.6602
learn: 0.6365
Learning: 0.6208
reteaching: 0.5810
learner_centered: 0.5739
```

9.Aim: Write a program to identify the words that are most common to a given word using similarity measures and pre-trained model representations.

Description:

This program identifies words most similar to a given input word using pre-trained word embeddings and similarity measures.

It utilizes the spaCy 'en_core_web_md' model, a medium-sized English language model with pre-trained word vectors.

The program computes cosine similarity between the input word and all words in the vocabulary, then retrieves the top N most similar words.

Pre-trained Model Used:
- 'en_core_web_md': A medium-sized model trained on a mix of web texts, containing word embeddings that capture semantic relationships.
- Can be replaced with 'en_core_web_lg' for higher accuracy.

Program:

```
import spacy
import numpy as np
from scipy.spatial.distance import cosine

def find_similar_words(word, top_n=10):
    nlp = spacy.load("en_core_web_md")
    word_doc = nlp(word)
    if not word_doc.vector.any():
        print(f"Word '{word}' is not in the vocabulary.")
        return []
    word_vector = word_doc.vector
    similarities = []
    for token in nlp.vocab:
        if token.has_vector and token.is_alpha and not token.is_stop:
            similarity = 1 - cosine(word_vector, token.vector)
            similarities.append((token.text, similarity))
    sorted_similarities = sorted(similarities, key=lambda x: x[1], reverse=True)
    return sorted_similarities[:top_n]
word = input("Enter a word: ")
similar_words = find_similar_words(word)
if similar_words:
    print(f"\nMost similar words to '{word}':")
    for w, score in similar_words:
        print(f"{w}: {score:.4f}")
```

Output:

```
Enter a word: happy
Most similar words to 'happy':
happy: 1.0000
lovin: 0.5306
somethin: 0.4649
nothin: 0.4189
need: 0.4143
Somethin: 0.4034
ought: 0.3973
let: 0.3801
Doin: 0.3224
Nothin: 0.3173
```

10.Aim: Write a program to generate new text using back translation.

Description:

Back translation is a data augmentation technique in Natural Language Processing (NLP) where a text is translated into another language and then translated back into the original language. This method helps generate paraphrased versions of text while preserving its meaning, improving model robustness, and enhancing training datasets for machine learning tasks like text classification, machine translation, and sentiment analysis.

Program:

```
import random
from deep_translator import GoogleTranslator
def back_translate(text, src_language='en', intermediate_languages=['fr', 'de', 'es','te']):
    translator = GoogleTranslator()
    intermediate_language = random.choice(intermediate_languages)
    translated = GoogleTranslator(source=src_language,
target=intermediate_language).translate(text)
    back_translated = GoogleTranslator(source=intermediate_language,
target=src_language).translate(translated)
    return translated, back_translated
text=input("enter text:")
intermediate, new_text = back_translate(text)
print("Original Text:", text)
print("Intermediate Translated Text:", intermediate)
print("Back Translated Text:", new_text)
```

Output:

```
enter text:Technology is evolving rapidly, changing the way we live and work."
Original Text: Technology is evolving rapidly, changing the way we live and work."
Intermediate Translated Text: La tecnología está evolucionando rápidamente, cambiando la
forma en que vivimos y trabajamos ".
Back Translated Text: Technology is quickly evolving, changing the way we live and work. "
```