ROUGH RECORD

AML 332 NATURAL LANGUAGE PROCESSING LAB

SEMESTER – VI

Submitted by

Name: ....
Roll No:..

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

MANGALAM
COLLEGE OF ENGINEERING
ETTUMANOOR, KOTTAYAM DIST.

विश्वं शास्त्रे प्रतिष्ठितम्।

Affiliated to APJ Abdul Kalam Technological University

Approved by the All India Council for Technical Education (AICTE)

MANGALAM COLLEGE OF ENGINEERING
DEPARTMENT OF ARTIFICIAL INTELLIGENCE & MACHINE

Vision

To become center of excellence in computing and research where future generations embrace technologies wholeheartedly and use its possibilities to make the world a better place to live.

Mission

Enlighten the young talents to achieve academic excellence as well as professional competence by imparting state of the art knowledge in computing and to be admirable individuals with ethical values and appropriate skills.

Program Educational Objectives

PEO 1: Graduate will have strong foundation and profound knowledge in computing and allied engineering and be able to analyze the requirements of real-world problems, design and develop innovative engineering solutions and maintain it effectively in the profession.

PEO 2: Graduate will adapt to technological advancements by engaging in higher studies, lifelong learning and research, there by contribute to computing profession.

PEO 3: Graduate will foster team spirit, leadership, communication, ethics and social values, which will lead to apply the knowledge of societal impacts of computing technologies.

Program Specific Outcomes

PSO 1: Apply the computing principles of artificial Intelligence and Machine Learning in the multi–disciplinary areas.

PSO 2: Demonstrate engineering knowledge in Artificial Intelligence and Machine Learning learned to solve real time problems in various domains

COURSE OUTCOMES

CO1	Apply the concept of natural language processing (NLP) using Natural Language Toolkit (NLTK).**(Cognitive Knowledge Level: Apply)**
CO2	Build text corpora with tokenization, Stemming, Lemmatization and apply visualization techniques.**(Cognitive Knowledge Level: Apply)**
CO3	Evaluate the classifiers and choose the best classifier. **(Cognitive Knowledge Level: Apply)**
CO4	Create Artificial Intelligence applications for text data. **(Cognitive Knowledge Level: Apply)**

Mapping of course outcomes with program outcomes

	PO1	**PO2**	**PO3**	**PO4**	**PO5**	**PO6**	**PO7**	**PO8**	**PO9**	**PO10**	**PO11**	**PO12**
CO1	2	2	-	-	-	-	-	-	-	-	-	-
CO2	2	2	-	-	-	-	-	-	-	-	-	-
CO3	2	2	1	-	-	-	-	-	-	-	-	-
CO4	2	2	1	-	-	-	-	-	-	-	-	-

INDEX

SL.N O.	NAME OF EXPERIMENT	DATE	SIGN
1	Familiarize With Python Natural Language Processing Toolkit Nltk		
2	Implement The Concept Of Concordances		
3	Program To Check For Word Count In A Given Text		
4	Python Program To Perform Preprocessing Of Text		
5	Program To Replace Words With Its Synonyms And Words Matching Regular Expressions.		
6	Program To Build Bag Of Words Model		
7	Program To Find Tf-Idf Values Of Each Words In A Document		
8	Implement Named Entity Recognition Using Nltk		
9	Program to rank the documents in a corpus using TF-IDF		
10	Implement Machine Learning based Text Classification in Python (Naive Bayes Classifier)		
11	Implement A Basic Chatbot Using Python		
12	Implement a language translator using Python		

NATURAL LANGUAGE PROCESSING

Natural Language Processing (NLP) is a subfield of computer science and artificial intelligence that focuses on the interaction between computers and human languages[12]. The main goal of NLP is to enable computers to understand, interpret, and generate natural language in a way that is similar to how humans do.

NLP involves a variety of techniques, including computational linguistics, machine learning, and statistical modeling[1]. It is used to analyze, understand, and generate natural language data, such as text and speech[1]. NLP can be applied to tasks such as sentiment analysis, machine translation, text classification, and more.

One of the challenges in NLP is dealing with the unstructured nature of human language. While computers can easily understand structured data like spreadsheets and database tables, human languages, texts, and voices form an unstructured category of data, making it difficult for computers to understand[1]. Despite these challenges, NLP has been successfully used in various applications, such as virtual assistants like Siri, Cortana, and Alexa, web search, email spam filtering, automatic translation of text or speech, document summarization, and grammar/spell checking.

In order to process and understand natural language, NLP often involves building a pipeline, which breaks down the complex problem of understanding language into smaller, more manageable tasks[1]. Each of these tasks is then modeled and solved individually, and their outputs are integrated to produce the final result.

Despite the complexity and challenges, NLP holds great promise in helping us leverage the vast amounts of natural language data available in the world, from literature and historical documents to social media posts and beyond.

There are several tools available for implementing Natural Language Processing (NLP). Here are some of the most popular ones:

NLTK (Natural Language Toolkit): A popular Python library for NLP tasks such as tokenization, stemming, tagging, parsing, and semantic reasoning.

spaCy: An industrial-strength NLP library that offers efficient tokenization, named entity recognition (NER), part-of-speech tagging, and dependency parsing.

Gensim: A Python library for topic modeling and document similarity analysis, including algorithms like Latent Semantic Analysis (LSA) and Latent Dirichlet Allocation (LDA).

The Natural Language Toolkit, or NLTK, is a leading platform for building Python programs to work with human language data1. It provides easy-to-use interfaces to over 50 corpora and lexical resources such as WordNet, along with a suite of text processing libraries for classification, tokenization, stemming, tagging, parsing, and semantic reasoning.

NLTK also includes wrappers for industrial-strength NLP libraries and has an active discussion forum1. It is suitable for linguists, engineers, students, educators, researchers, and industry users alike. NLTK is available for Windows, Mac OS X, and Linux. It is a free, open-source, community-driven project.

NLTK has been called "a wonderful tool for teaching, and working in, computational linguistics using Python," and "an amazing library to play with natural language1." The toolkit also provides a practical introduction to programming for language processing

Experiment Number: 1 **Date of Experiment:**

FAMILIARIZE WITH NLP TOOLKIT

AIM

Familiarize with python natural language processing toolkit nltk.

ALGORITHM

Getting Started with Python's NLTK

The first thing you need to do is make sure that you have Python installed. Once you have that dealt with, your next step is to install NLTK with pip.

```
$ python -m pip install nltk==3.5
```

Tokenizing:

By tokenizing, you can conveniently split up text by word or by sentence. This will allow you to work with smaller pieces of text that are still relatively coherent and meaningful even outside of the context of the rest of the text. It's your first step in turning unstructured data into structured data, which is easier to analyse.
When you're analyzing text, you'll be tokenizing by word and tokenizing by sentence. Here's what both types of tokenization bring to the table:

Tokenizing by word:

Words are like the atoms of natural language. They're the smallest unit of meaning that still makes sense on its own. Tokenizing your text by word allows you to identify words that come up particularly often. For example, if you were analysing a group of job ads, then you might find that the word "Python" comes up often. That could suggest high demand for Python knowledge, but you'd need to look deeper to know more.

Tokenizing by sentence:

When you tokenize by sentence, you can analyse how those words relate to one another and see more context. Are there a lot of negative words around the word "Python" because the hiring manager doesn't like Python? Are there more terms from the domain of herpetology than the domain of software development, suggesting that you may be dealing with an entirely different kind of python than you were expecting?
First we have to import the library "tokenize"

```
from nltk.tokenize import sent_tokenize, word_tokenize
sent_tokenize(example_string)
word_tokenize(example_string)
```

Filtering Stop Words:

Stop words are words that you want to ignore, so you filter them out of your text when you're processing it. Very common words like 'in', 'is', and 'an' are often used as stop words since they don't add a lot of meaning to a text in and of themselves.

```
nltk.download("stopwords")
from nltk.corpus import stopwords
stop_words = set(stopwords.words("english"))
filtered_list = []
for word in words_in_quote:
...    if word.casefold() not in stop_words:
...         filtered_list.append(word)
```

Stemming:

Stemming is a text processing task in which you reduce words to their root, which is the core part of a word. For example, the words “helping” and “helper” share the root “help.” Stemming allows you to zero in on the basic meaning of a word rather than all the details of how it’s being used. NLTK has more than one stemmer.

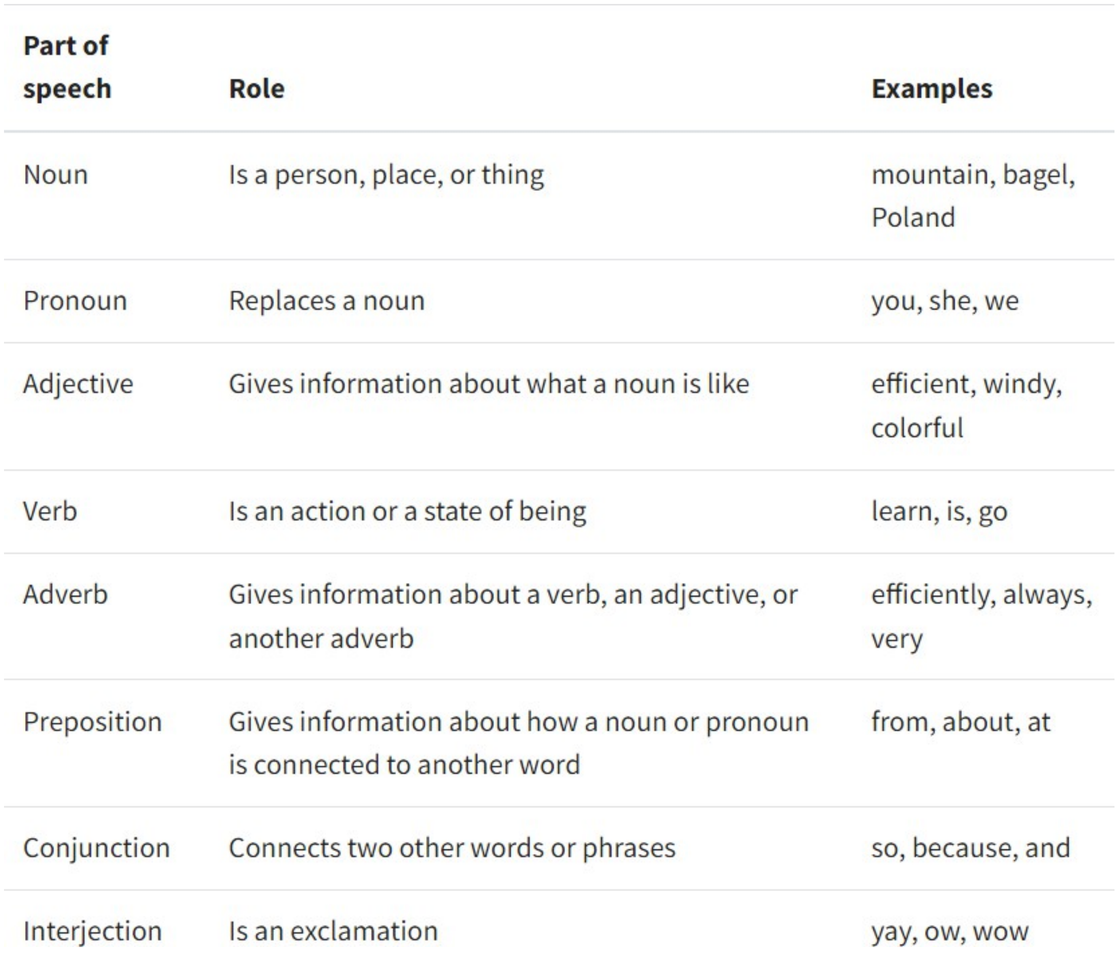
```
from nltk.stem import PorterStemmer
stemmer = PorterStemmer()
stemmed_words = [stemmer.stem(word) for word in words]
```

Understemming and overstemming are two ways stemming can go wrong:
Under stemming happens when two related words should be reduced to the same stem but aren’t. This is a false negative. Over stemming happens when two unrelated words are reduced to the same stem even though they shouldn’t be. This is a false positive.
The Porter stemming algorithm dates from 1979, so it’s a little on the older side. The Snowball stemmer, which is also called Porter2, is an improvement on the original and is also available through NLTK, so you can use that one in your own projects. It’s also worth noting that the purpose of the Porter stemmer is not to produce complete words but to find variant forms of a word.

Tagging Parts of Speech:

Part of speech is a grammatical term that deals with the roles words play when you use them together in sentences. Tagging parts of speech, or POS tagging, is the task of labeling the words in your text according to their part of speech.
In English, there are eight parts of speech:

```
import nltk
nltk.pos_tag(words_in_sagan_quote)


nltk.help.upenn_tagset() #To get a list of tags and their meanings
```



Part of speech	Role	Examples
Noun	Is a person, place, or thing	mountain, bagel, Poland
Pronoun	Replaces a noun	you, she, we
Adjective	Gives information about what a noun is like	efficient, windy, colorful
Verb	Is an action or a state of being	learn, is, go
Adverb	Gives information about a verb, an adjective, or another adverb	efficiently, always, very
Preposition	Gives information about how a noun or pronoun is connected to another word	from, about, at
Conjunction	Connects two other words or phrases	so, because, and
Interjection	Is an exclamation	yay, ow, wow

Lemmatizing:

Now that you're up to speed on parts of speech, you can circle back to lemmatizing. Like stemming, lemmatizing reduces words to their core meaning, but it will give you a complete English word that makes sense on its own instead of just a fragment of a word like 'discoveri'.

Note: A lemma is a word that represents a whole group of words, and that group of words is called a lexeme. For example, if you were to look up the word "blending" in a dictionary, then you'd need to look at the entry for "blend," but you would find "blending" listed in that entry.
In this example, "blend" is the lemma, and "blending" is part of the lexeme. So when you lemmatize a word, you are reducing it to its lemma

```
from nltk.stem import WordNetLemmatizer
lemmatizer = WordNetLemmatizer()
lemmatizer.lemmatize("scarves")
lemmatized_words = [lemmatizer.lemmatize(word) for word in words]
```

Chunking:

While tokenizing allows you to identify words and sentences, chunking allows you to identify phrases.
Note: A phrase is a word or group of words that works as a single unit to perform a grammatical function. Noun

phrases are built around a noun.

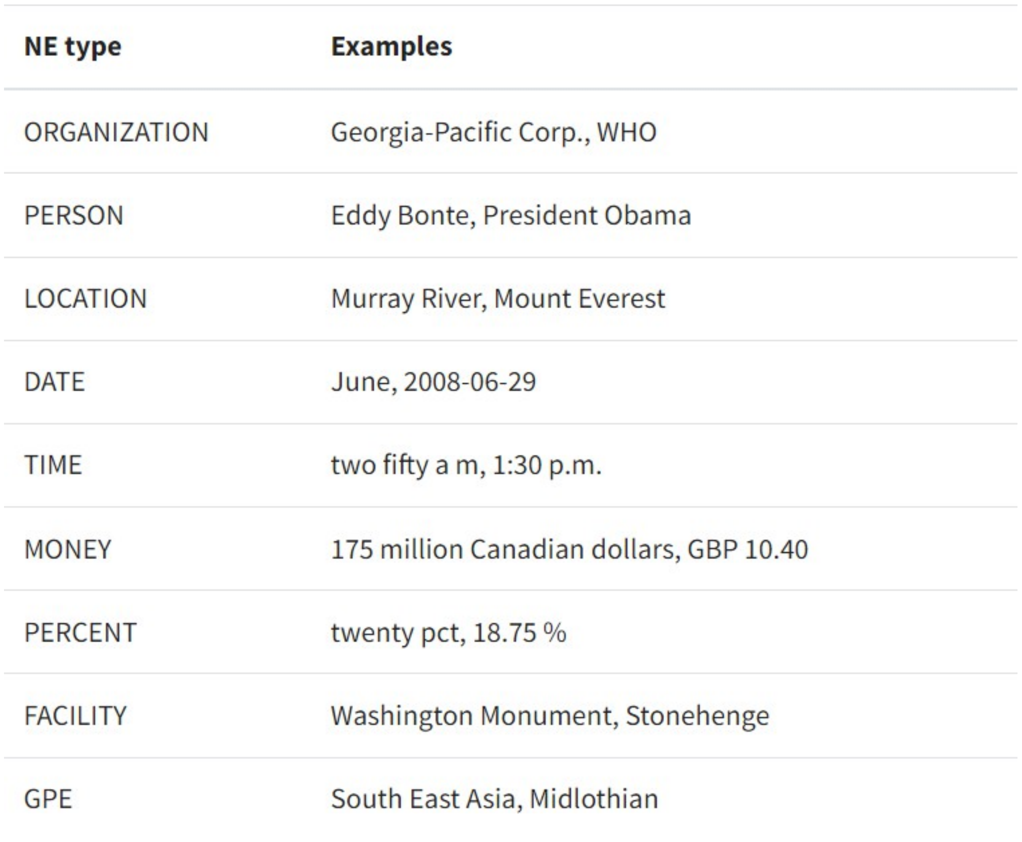
Here are some examples:
“A planet”
“A tilting planet”
“A swiftly tilting planet”

Chunking makes use of POS tags to group words and apply chunk tags to those groups. Chunks don’t overlap, so one instance of a word can be in only one chunk at a time.
In order to chunk, you first need to define a chunk grammar.
Note: A chunk grammar is a combination of rules on how sentences should be chunked. It often uses regular expressions, or regexes.
For this tutorial, you don’t need to know how regular expressions work, but they will definitely come in handy for you in the future if you want to process text.

```
grammar = "NP: {<DT>?<JJ>*<NN>}"

chunk_parser = nltk.RegexpParser(grammar)

tree = chunk_parser.parse(lotr_pos_tags)

tree.draw()
```

Using Named Entity Recognition (NER):

Named entities are noun phrases that refer to specific locations, people, organizations, and so on. With named entity recognition, you can find the named entities in your texts and also determine what kind of named entity they are.



NE type	Examples
ORGANIZATION	Georgia-Pacific Corp., WHO
PERSON	Eddy Bonte, President Obama
LOCATION	Murray River, Mount Everest
DATE	June, 2008-06-29
TIME	two fifty a m, 1:30 p.m.
MONEY	175 million Canadian dollars, GBP 10.40
PERCENT	twenty pct, 18.75 %
FACILITY	Washington Monument, Stonehenge
GPE	South East Asia, Midlothian

```
def extract_ne(quote):
```

```
...     words = word_tokenize(quote, language=language)
...     tags = nltk.pos_tag(words)
...     tree = nltk.ne_chunk(tags, binary=True)
...     return set(
...         " ".join(i[0] for i in t)
...         for t in tree
...         if hasattr(t, "label") and t.label() == "NE"
...     )
```

With this function, you gather all named entities, with no repeats. In order to do that, you tokenize by word, apply part of speech tags to those words, and then extract named entities based on those tags. Because you included binary=True, the named entities you'll get won't be labeled more specifically. You'll just know that they're named entities.

Making a Dispersion Plot:

You can use a dispersion plot to see how much a particular word appears and where it appears. So far, we've looked for "man" and "woman", but it would be interesting to see how much those words are used compared to their synonyms:

```
text8.dispersion_plot(["woman", "lady", "girl", "gal", "man", "gentleman", "boy", "guy"]
... )
```

Making a Frequency Distribution:

With a frequency distribution, you can check which words show up most frequently in your text.

```
from nltk import FreqDist
frequency_distribution = FreqDist(text8)
frequency_distribution.plot(20,  cumulative=True)
```

RESULT

Different functions of nltk toolkit is familiarized.

Experiment Number: 2 **Date of Experiment:**

CONCORDANCES

AIM

Choose an English word, and see how it is used in the different example texts by making concordances.

ALGORITHM

1. Start

2. Acquire Resources: Download or locate a collection of text data (e.g., the Gutenberg corpus)

3. Obtain Specific Text: Within the chosen corpus, select the text you want to analyze (e.g., "Moby Dick" by Melville)

4. Create a Text Object: Construct a representation of the text that facilitates analysis, providing convenient methods for exploration

5. Generate Concordance: Specify the target word. Indicate the desired amount of context to display (e.g.: 70 characters on either side)

6. Stop

PROGRAM CODE

```
pip install nltk

import nltk

from nltk.corpus import gutenberg

from nltk.text import Text

nltk.download('gutenberg')

text1 = gutenberg.words('melville-moby_dick.txt')

text2 = Text(text1)

concordance = text2.concordance('freedom', width=70)

concordance
```

or

```
import nltk
from nltk.tokenize import word_tokenize

example_string = “Now that you’re up to speed on parts of speech, you can circle back to
lemmatizing. Like stemming, lemmatizing reduces words to their core meaning”

tokens = word_tokenize(example_string)

conc = nltk. concordanceIndex(tokens)

conc. print_concordance(‘speed’)
```

OUTPUT:

```
Displaying 2 of 2 matches:

 that Gabriel had the complete freedom of the ship . The consequence

ination . See with what entire freedom the whaleman takes his handful
```

RESULT

The python program for implementing concordances is executed successfully.

Experiment Number: 3 **Date of Experiment:**

COUNTING VOCABULARY

AIM

Counting Vocabulary:

1. How many words (tokens) are there in the given text.
2. How many different words (types) are there in the given text
3. How many times does the word the occur in the text
4. What is this as a percentage of all the words in the text?

ALGORITHM

1.

1. Start.
2. Obtain Text
3. Prepare Text Processing Tools: Acquire a suitable library or toolkit capable of handling text manipulation
4. Split Text into Words: Employ a "tokenizer" function from the chosen library to divide the text into individual words (tokens).
5. Count the Tokens
6. Present the Result
7. Stop.

2.

1. Start.
2. Obtain Text
3. Prepare Text Processing Tools: Acquire a suitable library or toolkit capable of handling text manipulation
4. Split Text into Words: Employ a "tokenizer" function from the chosen library to divide the text into individual words (tokens).
5. Create Set of Unique Words.

6. Count the Tokens.
7. Present the Result.
8. Stop.

3.

1. Start
2. Prepare the Text
3. Split the Text into Words: Employ a "tokenizer" function to divide the text into individual words (tokens).
4. Count the Occurrences of the Target the Word: Use a counting function to determine how many times the target word appears within the list of tokens.
5. Present the Result
6. Stop

4.

1. Start

2. Gather Text and Word:

3. Split Text into Words: Use a "tokenizer" function to divide the text into individual words (tokens).

4. Count Word Occurrences

5. Count Total Words: Calculate the total number of words in the text.

6. Calculate Percentage: Divide the word count by the total word count and multiply by 100 to obtain the percentage.

7. Present Results

8. Stop

PROGRAM CODE:

1.

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

text = "This is a sample text to count the number of words. It contains multiple sentences and
punctuation."

words = nltk.word_tokenize(text)

num_words = len(words)

print("The number of words (tokens) in the text is:", num_words)
```

2.

```
import nltk

text = "This is a sample text with some repeated words to demonstrate counting unique words."

words = nltk.word_tokenize(text)

unique_words = set(words)

num_unique_words = len(unique_words)

print("The number of unique words (types) in the text is:", num_unique_words)
```

3.

```
import nltk

text = "This is the text we'll use to count how many times the word 'the' appears. The more the
merrier!"

words = nltk.word_tokenize(text)

count = words.count("the")

print("The word 'the' appears", count, "times in the text.")
```

4.

```
import nltk

text = "This is the text we'll use to count how many times the word 'the' appears. The more the
```

```
merrier!"

target_word = "the"

words = nltk.word_tokenize(text)

word_count = words.count(target_word)

total_words = len(words)

percentage = (word_count / total_words) * 100

print("The word '" + target_word + "' appears", word_count, "times in the text.")
print("This is", percentage, "% of all the words in the text.")
```

OUTPUT:

1.

```
The number of words (tokens) in the text is: 19
```

2.

```
The number of unique words (types) in the text is: 14
```

3.

```
The word 'the' appears 3 times in the text.
```

4.

```
The word 'the' appears 3 times in the text.
This is 13.043478260869565 % of all the words in the text.
```

RESULT

The python program for counting vocabulary is executed successfully.

Experiment Number: 4 **Date of Experiment:**

PREPROCESSING OF TEXT

AIM

Write Python program to perform preprocessing of text (Tokenization, Filtration, Stop Word Removal, Stemming and Lematization).

ALGORITHM

1. Start.
2. Input: Get the text.
3. Tokenize: Split text into tokens.
4. Remove Stop Words: Discard stop words.(Build In stop words and Custom Stopwords)
5. Stem: Apply stemming algorithm.
6. Lemma: Apply lemmatization algorithm.
7. Return: Return preprocessed tokens
8. Stop.

PROGRAM CODE:

```
import nltk

nltk.download('punkt')  # For tokenization
nltk.download('stopwords')  # For stop word removal

text = "This is a sample text to demonstrate text preprocessing techniques."

# 1. Tokenization
 tokens = nltk.word_tokenize(text)

print("Tokenized words", tokens)

# 2. Stop Word Removal
stop_words = set(nltk.corpus.stopwords.words('english'))

stop_removed = []

for i in tokens:
        if (i not in stop_words):
                stop_removed.append(i)
```

```
print("Build In Stop words removed", stop_removed)

custom_stop = ["text", "sample"]

stop_words.extend(custom_stop)

stop_removed_new = []


for i in tokens:
        if (i not in stop_words):
                stop_removed_new.append(i)


print("Custom Stop words removed", stop_removed_new)


# 3. Stemming


stem_removed = []

stemmer = nltk.PorterStemmer()

for token in stopremoved:
        stem_removed.append(stemmer.stem(token))


# 3. Lemmatization

import nltk
from nltk.corpus import wordnet

nltk.download('wordnet')
nltk.download('averaged_perceptron_tagger')

text = "Running is fun and playing games is enjoyable."

lemmatizer = wordnet.WordNetLemmatizer()

pos_tags = nltk.pos_tag(tokens)

lemmatized_words = ""

for word, pos_tag in pos_tags:

          if pos_tag.startswith('V'):  # For verbs
                pos='v'

          elif pos_tag.startswith('N'):  # For nouns
                pos='n'
        elif pos_tag.startswith('J'):  # For adjectives
            pos='a'

        elif pos_tag.startswith('R'):  # For adverbs
            pos='r'

        else:
            lemmatized_words  = lemmatized_words  + lemmatized_words.append(word) + " "

    lemmatized_words = lemmatized_words + lemmatizer.lemmatize(word, pos=pos) + " "

print(f"Original text: {text}")
print(f"\nLemmatized text: { lemmatized_words }")
```

OUTPUT:

```
Tokenized words ['This', 'is', 'a', 'sample', 'text', 'to', 'demonstrate', 'text', 'preprocessing', 'techniques',
'.']


Build In Stop words removed ['This', 'sample', 'text', 'demonstrate', 'text', 'preprocessing', 'techniques',
'.']

Custom Stop words removed ['This', 'demonstrate', 'preprocessing', 'techniques', '.']


Original text: This is a sample text to demonstrate text preprocessing techniques.

Stemmed text: ['thi', 'sampl', 'text', 'demonstr', 'text', 'preprocess', 'techniqu', '.']


Original text: Running is fun and playing games is enjoyable.

Lemmatized text: This a to . .
```

RESULT:

The Python program to perform preprocessing of text has been executed successfully.

Experiment Number: 5 **Date of Experiment:**

REPLACE WORDS WITH ITS SYNONYMS AND WORDS MATCHING REGULAR EXPRESSIONS.

AIM

Write a python program to replace words with its synonyms and words matching regular expressions.

ALGORITHM

Replace words with its synonyms:

1. Start.
2. Tokenization - Split the input sentence into individual words using nltk.word_tokenize.
3. Synonym Retrieval - Use wordnet.synsets to find all synonyms (synsets) of the target word
4. Sentence Reconstruction - Create an empty list named new_sentence to store the modified sentence.
5. Return Modified Sentence
6. Stop.

Replace words matching regular expressions:

1. Start
2. Define the replacement function
3. Apply the replacement - use the re.sub() function to replace occurrences of the pattern in the
4. Set the text and pattern - Define the regular expression pattern (regex) that specifies the text you want to replace.
5. Set the replacement string
6. Call the replacement function
7. Print the original and modified text
8. Stop

PROGRAM CODE

```
#Replace words with its synonyms:

import nltk
from nltk.corpus import wordnet
import random

sentence = "The quick brown fox jumps over the lazy dog."
```

```
word = "quick"

print("Original Sentence: ", sentence)

tokens = nltk.word_tokenize(sentence)

synonyms = []

  for synset in wordnet.synsets(word):
    for synonym in synset.lemmas():
      synonyms.append(synonym.name())

  new_sentence = ""

  for i in tokens:
    if word == i:
      new_sentence = new_sentence + synonyms[random.randint(0,len(synonyms))] + " "
    else:
      new_sentence = new_sentence + i + " "

print("New Sentence: ", new_sentence)


#Replace words matching regular expressions:

import re

text = "This is a sample text with some words to be replaced. Another line with words."

regex = r"\b(sample)\b"

replacement = "hello"

new_text = re.sub(regex, replacement, text)

print(f"Original text:\n{text}")
print(f"\nModified text:\n{new_text}")
```

OUTPUT

```
#Replace words with its synonyms:

Original Sentence: The quick brown fox jumps over the lazy dog.

New Sentence: The spry brown fox jumps over the lazy dog

#Replace words matching regular expressions:
```

```
Original Text: This is a sample text with some words to be replaced. Another line with words
Modified Text:  This is a hello text with some words to be replaced. Another line with words
```

RESULT:

Python program to replace words with its synonyms and words matching regular expressions have been executed successfully.

Experiment Number: 6 **Date of Experiment:**

PYTHON PROGRAM TO BUILD BAG OF WORDS MODEL.

AIM

Write a python program to build bag of words model.

ALGORITHM

1. Start.
2. Initialize a CountVectorizer object:
3. Fit the vectorizer to the documents:
4. Iterate through each document: For each document, transform the document
5. Use the fitted vectorizer to convert the document into a numerical vector representing the frequency of each word in the vocabulary.
6. Print the document
7. Stop.

PROGRAM CODE

```
from sklearn.feature_extraction.text import CountVectorizer

documents = [
    "This is the first this document.",
    "This is the second document with some overlap.",
    "This is the third document with new information."
]

# Create the Bag of Words model
vectorizer = CountVectorizer()

# Fit the model on the preprocessed documents
bow_model = vectorizer.fit_transform(documents)

# Print the vocabulary
print("Vocabulary:", vectorizer.get_feature_names_out())

# Print the Bag of Words representation for each document
print("Bag of Words:")

for i in range(0,len(documents)):
    print(f"{documents[i]}:\n {bow_model.toarray()[i]}")
```

OUTPUT

```
Vocabulary: ['document', 'first', 'information', 'is', 'new', 'overlap', 'second', 'some', 'the', 'third', 'this',
'with']

Bag of Words:

This is the first this document.:

 [1 1 0 1 0 0 0 0 1 0 2 0]

This is the second document with some overlap.:

 [1 0 0 1 0 1 1 1 1 0 1 1]

This is the third document with new information.:

 [1 0 1 1 1 0 0 0 1 1 1 1]
```

RESULT:

Python program to build bag of words model have been executed successfully.

Experiment Number: 7 **Date of Experiment:**

PYTHON PROGRAM TO FIND TF-IDF VALUES OF EACH WORDS IN A DOCUMENT

AIM

Write A Python Program To Find Tf-Idf Values Of Each Words In A Document

ALGORITHM

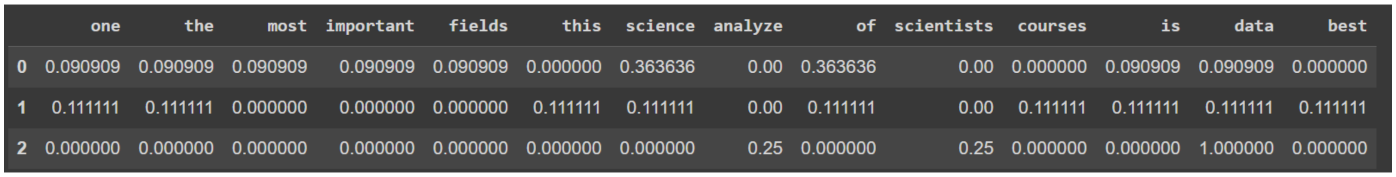
1. Start.
2. Create a Vocabulary
3. Preprocess the data
4. Calculate Term Frequency: number of occurrences of the word divided by the total number of words in the document
5. Calculate Document Frequency (DF) : Initialize a counter k to 0. Iterate through each document and if the word exists in the tokenized document, increment k
6. Calculate Inverse Document Frequency (IDF): Calculate IDF using the formula: idf[w] = log10(number of documents / k).
7. Calculate TF-IDF : Multiply TF value with the corresponding IDF value
8. Stop.

PROGRAM CODE

```
import pandas as pd
import numpy as np
import nltk

nltk.download('punkt')

corpus = ['data science is one of the most important fields of science',
         'this is one of the best data science courses',
         'data scientists analyze data']
words_set = set()

for doc in  corpus:
    words = nltk.word_tokenize(doc.lower())
    words_set = words_set.union(set(words))

n_docs = len(corpus)         #·Number of documents in the corpus
n_words_set = len(words_set) #·Number of unique words in the
```

```
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 = nltk.word_tokenize(corpus[i].lower())
    # words = corpus[i].split(' ') # Words in the document
    for w in words:
        df_tf[w][i] = df_tf[w][i] + (words.count(w) / len(words))

print(df_tf)

print("IDF of: ")

idf = {}

for w in words_set:
    k = 0    # number of documents in the corpus that contain this word

    for i in range(n_docs):
        if w in nltk.word_tokenize(corpus[i].lower()):
            k += 1

    idf[w] =  np.log10(n_docs / k)

    print(w, " : ", idf[w] )

df_tf_idf = df_tf.copy()
for w in words_set:
    for i in range(n_docs):
        df_tf_idf[w][i] = df_tf[w][i] * idf[w]

print(df_tf_idf)
```

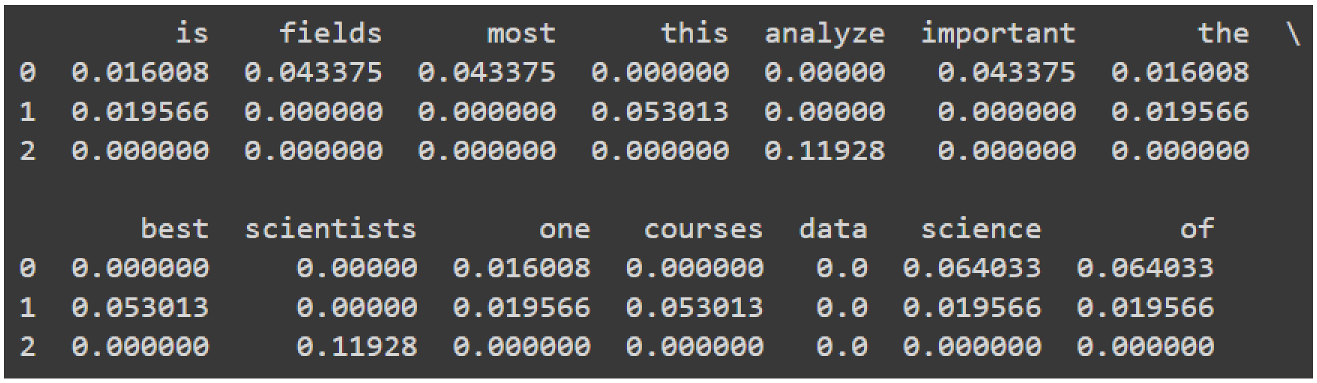
OUTPUT



	one	the	most	important	fields	this	science	analyze	of	scientists	courses	is	data	best
0	0.090909	0.090909	0.090909	0.090909	0.090909	0.000000	0.363636	0.00	0.363636	0.00	0.000000	0.090909	0.090909	0.000000
1	0.111111	0.111111	0.000000	0.000000	0.000000	0.111111	0.111111	0.00	0.111111	0.00	0.111111	0.111111	0.111111	0.111111
2	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.25	0.000000	0.25	0.000000	0.000000	1.000000	0.000000

```
IDF of:

one  :  0.17609125905568124

the  :  0.17609125905568124

most  :  0.47712125471966244
```

```
important  :  0.47712125471966244

fields  :  0.47712125471966244

this  :  0.47712125471966244

science  :  0.17609125905568124

analyze  :  0.47712125471966244

of  :  0.17609125905568124

scientists  :  0.47712125471966244

courses  :  0.47712125471966244

is  :  0.17609125905568124

data  :  0.0

best  :  0.47712125471966244
```



```
         is    fields      most      this  analyze  important       the  \
0  0.016008  0.043375  0.043375  0.000000  0.00000   0.043375  0.016008   
1  0.019566  0.000000  0.000000  0.053013  0.00000   0.000000  0.019566   
2  0.000000  0.000000  0.000000  0.000000  0.11928   0.000000  0.000000   

       best  scientists       one   courses  data   science        of  
0  0.000000     0.00000  0.016008  0.000000   0.0  0.064033  0.064033  
1  0.053013     0.00000  0.019566  0.053013   0.0  0.019566  0.019566  
2  0.000000     0.11928  0.000000  0.000000   0.0  0.000000  0.000000  
```

RESULT:

Python program to Find Tf-Idf Values Of Each Words In A Document have been executed successfully.

Experiment Number: 8 **Date of Experiment:**

IMPLEMENT NAMED ENTITY RECOGNITION USING NLTK

AIM

Write A Python Program To Implement Named Entity Recognition Using Nltk

ALGORITHM

1. Start.
2. Define the text string (text) you want to analyze for named entities
3. Split the text into individual tokens (words)
4. Use pos_tag to assign part-of-speech tags to each token
5. Use ne_chunk to perform named entity recognition on the tokens
6. Print Entities
7. Stop.

PROGRAM CODE

```
nltk.download('averaged_perceptron_tagger')
nltk.download('maxent_ne_chunker')
nltk.download('words')

import nltk
from nltk.tokenize import word_tokenize
from nltk.tag import pos_tag
from nltk.chunk import ne_chunk

# Define the text to be analyzed
text = "GeeksforGeeks is a recognised platform for online learning in India"

# Tokenize the text
tokens = nltk.word_tokenize(text)

# Perform part-of-speech tagging
tagged = nltk.pos_tag(tokens)

# Perform named entity recognition
entities = nltk.chunk.ne_chunk(tagged)

# Print the named entities
print(entities)
```

OUTPUT

```
(S
  (ORGANIZATION GeeksforGeeks/NNP)
  is/VBZ
  a/DT
  recognised/JJ
  platform/NN
  for/IN
  online/NN
  learning/NN
  in/IN
  (GPE India/NNP))
```

RESULT:

Python program to Implement Named Entity Recognition Using Nltk have been executed successfully.

Experiment Number: 9 Date of Experiment:

THE MOST SIMILAR SENTENCE IN THE FILE TO THE GIVEN INPUT SENTENCE

AIM

Write A Python Program To Find The Most Similar Sentence In The File To The Given Input Sentence

ALGORITHM

1. Start.
2. Create a Vocabulary
3. Preprocess the data
4. Calculate Term Frequency: log10(number of occurrences of the word plus one)
5. Calculate Document Frequency (DF) : Initialize a counter k to 0. Iterate through each document and if the word exists in the tokenized document, increment k
6. Calculate Inverse Document Frequency (IDF): Calculate IDF using the formula: idf[w] = log10(number of documents / k).
7. Calculate TF-IDF : Multiply TF value with the corresponding IDF value. Calculate document length (|d|) as the square root of individual tf-idf value
8. Query Processing: Get the user query as input. Tokenize the query using NLTK
9. Document Scoring: Calculate score as sum of tf-idf values of each tokens in user query divided by document length (|d|)
10. Ranking: Add a column Rank for ranking documents based on their score in descending order.
11. Stop.

PROGRAM CODE

```
import pandas as pd
import numpy as np
import nltk

nltk.download('punkt')

corpus = ['data science is one of the most important fields of science',
        'this is one of the best data science courses',
        'data scientists analyze data' ]
```

```
words_set = set()

for doc in  corpus:
    words = nltk.word_tokenize(doc.lower())
    words_set = words_set.union(set(words))

n_docs = len(corpus)         #·Number of documents in the corpus
n_words_set = len(words_set) #·Number of unique words in the

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 = nltk.word_tokenize(corpus[i].lower())
    # words = corpus[i].split(' ') # Words in the document
    for w in words:
        df_tf[w][i] = df_tf[w][i] + (np.log10(words.count(w)))

idf = {}

for w in words_set:
    k = 0    # number of documents in the corpus that contain this word

    for i in range(n_docs):
        if w in nltk.word_tokenize(corpus[i].lower()):
            k += 1

    idf[w] =  np.log10(n_docs / k)

 df_tf_idf = pd.DataFrame(np.zeros((n_docs, n_words_set+1)), columns=list(words_set) + ['|d|'])

for i in range(n_docs):
  temp = 0

  for w in words_set:
    df_tf_idf[w][i] = df_tf[w][i] * idf[w]
    temp+= (df_tf_idf[w][i] * df_tf_idf[w][i])

  df_tf_idf['|d|'][i] = np.sqrt(temp)


print("Scoring the Documents: ")
query = str(input("Enter the query: "))
token_query = nltk.word_tokenize(query.lower())
score = pd.DataFrame(np.zeros((n_docs, len(token_query)+3)), columns = list(token_query) + ['|d|',
'Score', 'Rank'])

for i in range(n_docs):
  temp = 0
  for w in token_query:
    score[w][i] = score[w][i] + df_tf_idf[w][i]
    temp += score[w][i]

  score['|d|'][i] = df_tf_idf['|d|'][i]
  score['Score'][i] = round(temp/score['|d|'][i], 3)

score['Rank'] = score['Score'].rank(ascending=False)
score = score.sort_values(by='Rank')

print(score)
```

OUTPUT

```
Scoring the Documents:

Enter the query: data science

     data   science     |d|        Score    Rank

0    0.0    0.064033    0.120886   0.530    1.0

1    0.0    0.019566    0.101712   0.192    2.0

2    0.0    0.000000    0.168688   0.000    3.0
```

RESULT:

Python program to Find The Most Similar Sentence In The File To The Given Input Sentence have been executed successfully.

Experiment Number: **10** **Date of Experiment:**

MACHINE LEARNING BASED TEXT CLASSIFICATION IN PYTHON (NAIVE BAYES CLASSIFIER)

AIM

Implement Machine Learning based Text Classification in Python (Naive Bayes Classifier)

ALGORITHM

1. Start.
2. Data Preparation
3. Preprocess the data
4. Feature Extraction: Use a TF-IDF vectorizer to convert text data into numerical features. Fit the vectorizer to the training data to learn the vocabulary
5. Model Training: Create a Multinomial Naive Bayes classifier object and Train the classifier on the extracted training features and labels
6. Model Evaluation: Evaluate the model's performance on the test data using accuracy score.
7. Sentiment Prediction: Define a new piece of text to analyze its sentiment. Use the trained classifier to predict the sentiment of the new text.
8. Stop.

PROGRAM CODE

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.model_selection import train_test_split
import pandas as pd
import nltk
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer
import re

# Define stopwords and stemmer
nltk.download('stopwords')
stop_words = stopwords.words('english')
stemmer = PorterStemmer()


Text = pd.read_csv('/content/drive/My Drive/IMDB Dataset.csv')
```

```
documents = Text['review'].tolist()
labels = [0 if label == "negative" else 1 for label in Text['sentiment'].tolist()]

def preprocess_text(text):

  # Lowercase text
  text = text.lower()

  clean = re.compile('<.*?>')
  text = re.sub(clean, '', text)

  # Remove stopwords
  text = ' '.join([word for word in text.split() if word not in stop_words])

  # Perform stemming (optional)
  text = ' '.join([stemmer.stem(word) for word in text.split()])

  return text

pre_doc = [preprocess_text(doc) for doc in documents]

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(documents, labels, test_size=0.2)

# Feature extraction using TF-IDF vectorizer
vectorizer = TfidfVectorizer()
X_train_features = vectorizer.fit_transform(X_train)
X_test_features = vectorizer.transform(X_test)

# Train a Naive Bayes classifier
classifier = MultinomialNB()
classifier.fit(X_train_features, y_train)

# Evaluate performance on test data (optional)
from sklearn.metrics import accuracy_score
accuracy = accuracy_score(y_test, classifier.predict(X_test_features))
print("Accuracy:", accuracy)

# Predict sentiment on new text
new_text = "This movie was awesome, definitely recommend!"
new_text_features = vectorizer.transform([new_text])
predicted_sentiment = classifier.predict(new_text_features)[0]

# Print results
if predicted_sentiment == 1:
print("Predicted sentiment:", "positive")
else:
  print("Predicted sentiment:", “negative")
```

OUTPUT

```
Accuracy: 0.8

Predicted sentiment: positive
```

RESULT:

Python program to Implement Machine Learning based Text Classification in Python (Naive Bayes Classifier) have been executed successfully.

Experiment Number: **11** **Date of Experiment:**

IMPLEMENT A BASIC CHATBOT USING PYTHON

AIM

Write A Python Program to Implement a basic chatbot

ALGORITHM

1. Start.
2. Import Libraries
3. Define Patterns and Responses: Create a list of pairs, where each pair consists of: A pattern (regular expression) that captures user input and a list of corresponding responses that the chatbot can provide.
4. Initialize Chatbot: Create a Chat object using the defined patterns and reflections.
5. Start Chatbot Conversation: Call the converse method of the Chat object.
6. Stop.

PROGRAM CODE

```
# Import necessary modules
from nltk.chat.util import Chat, reflections

# Define pairs of patterns and corresponding responses
pairs = [
    ['my name is (.*)', ['Hello %1!']],
    ['(hi|hello|hey|holla|hola)', ['Hey there!', 'Hi there!', 'Hey!']],
    ['(.*) your name ?', ['My name is Bot.']],
    ['(.*) do you do ?', ['I am a chatbot.']],
    ['(.*) created you ?', ['I was created using Python and NLTK.']],
    ['bye', ['Goodbye! See you later.']]
]

# Initialize the chatbot
chat = Chat(pairs, reflections)

# Start the chatbot
chat.converse()
```

OUTPUT

```
>Hey
Hi there!
>Whats your name?
```

```
My name is Bot.
>My name is ABC
Hello abc!
>How do you do?
I am a chatbot.
>Who created you?
I was created using Python and NLTK.
```

RESULT:

Python program to Implement a basic chatbot have been executed successfully.

Experiment Number: **12** **Date of Experiment:**

IMPLEMENT A LANGUAGE TRANSLATOR

AIM

Write A Python Program to Implement a language translator

ALGORITHM

1. Start.
2. Data Loading and Preprocessing: Read the CSV file containing English and Spanish sentences using pandas. Define a function clean_sentences to perform basic text cleaning:
3. Training the Translation Model: Create a function train_translation_model to train the IBM Model 1. Call the train_translation_model function with the cleaned English and Spanish sentences to train the translation model.
4. Translating Input Sentences: Create a function translate_input to handle user interaction and translation
5. Stop.

PROGRAM CODE

```
import pandas as pd
import re
import nltk
from nltk.translate import AlignedSent, IBMModel1
# Load and Preprocess the Data
df = pd.read_csv("engspn.csv")
english_sentences = df['english'].tolist()
spanish_sentences = df['spanish'].tolist()

def clean_sentences(sentences):
    cleaned_sentences = []
    for sentence in sentences:
        sentence = sentence.strip()
        sentence = sentence.lower()
        sentence = re.sub(r"[^a-zA-Z0-9]+", " ", sentence)
        cleaned_sentences.append(sentence.strip())
    return cleaned_sentences

cleaned_english_sentences = clean_sentences(english_sentences)
cleaned_spanish_sentences = clean_sentences(spanish_sentences)

# Train the Translation Model
```

```
def train_translation_model(source_sentences, target_sentences):
    aligned_sentences = [AlignedSent(source.split(), target.split()) for source, target in
zip(source_sentences, target_sentences)]
    ibm_model = IBMModel1(aligned_sentences, 10)
    return ibm_model

translation_model = train_translation_model(cleaned_english_sentences, cleaned_spanish_sentences)

# Translate Input Sentences
def translate_input(ibm_model):
    while True:
        source_text = input("Enter the English sentence to translate (or 'q' to quit): ")
        if source_text.lower() == 'q':
            print("Quitting...")
            break
        cleaned_text = clean_sentences(source_text.split())
        source_words = cleaned_text
        translated_words = []
        for source_word in source_words:
            max_prob = 0.0
            translated_word = None
            for target_word in ibm_model.translation_table[source_word]:
                prob = ibm_model.translation_table[source_word][target_word]
                if prob > max_prob:
                    max_prob = prob
                    translated_word = target_word
            if translated_word is not None:
                translated_words.append(translated_word)
        translated_text = ' '.join(translated_words)
        print("Translated text:", translated_text)
        print()

translate_input(translation_model)
```

OUTPUT

```
Enter the English sentence to translate (or 'q' to quit): hello how are you?
Translated text: hello mo andan vosotras

Enter the English sentence to translate (or 'q' to quit): q
Quitting...
```

RESULT:

Python program to Implement a language translator have been executed successfully.