**Parle Tilak Vidyalaya Associations**

**M. L. DAHANUKAR COLLEGE**

**Vile-Parle (East), Mumbai – 400 057.**

**Practical Journal**

**NATURAL LANGUAGE PROCESSING**

**Submitted by**

**Trupti Hiru Bhostekar**

**Seat No.:**

**M.Sc. [I.T.]-Information Technology Part II**

**2023 – 2024**

**INDEX**

|  |  |  |  |
| --- | --- | --- | --- |
| **Sr no** | **AIM** | **Date** | **Sign** |
| **1** | 1. **Install NLTK** 2. **Convert the given text to speech** 3. **Convert audio file Speech to Text.** |  |  |
| **2** | 1. **Study of various Corpus – Brown, Inaugural, Reuters, udhr with various methods like fields, raw, words, sents, categories,** 2. **Create and use your own corpora(plaintext, categorical)** 3. **Study Conditional frequency distributions Study of tagged corpora with methods like tagged\_sents, tagged\_words.** 4. **Write a program to find the most frequent noun tags.** 5. **Map Words to Properties Using Python Dictionaries** 6. **Study DefaultTagger, Regular expression tagger, UnigramTagger** 7. **Find different words from a given plain text without any space by comparing this text with a given corpus of words. Also find the score of words.** |  |  |
| **3** | 1. **Study of Wordnet Dictionary with methods as synsets, definitions, examples, antonyms.** 2. **Study lemmas, hyponyms, hypernyms, entailments,** 3. **Write a program using python to find synonym and antonym of word "active" using Wordnet** 4. **Compare two nouns** 5. **Handling stopword.**   **Using nltk Adding or Removing Stop Words in NLTK's Default Stop Word List**  **Using Gensim Adding and Removing Stop Words in Default Gensim Stop Words List**  **Using Spacy Adding and Removing Stop Words in Default Spacy Stop Words List** |  |  |
| **4** | **Text Tokenization**   1. **Tokenization using Python’s split() function** 2. **Tokenization using Regular Expressions (RegEx)** 3. **Tokenization using NLTK** 4. **Tokenization using the spaCy library** 5. **Tokenization using Keras** 6. **Tokenization using Gensim** |  |  |
| **5** | **Important NLP Libraries for Indian Languages and perform:**   1. **word tokenization in Hindi** 2. **Generate similar sentences from a given Hindi text input** 3. **Identify the Indian language of a text** |  |  |
| **6** | **Illustrate part of speech tagging.**   1. **Part of speech Tagging and chunking of user defined text.** 2. **Named Entity recognition of user defined text.** 3. **Named Entity recognition with diagram using NLTK corpus – treebank** |  |  |
| **7** | 1. **Define grammer using nltk. Analyze a sentence using the same.** 2. **Accept the input string with Regular expression of FA: 101+** 3. **Accept the input string with Regular expression of FA: (a+b)\*bba** 4. **Implementation of Deductive Chart Parsing using context free grammar and a given sentence.** |  |  |
| **8** | **Study PorterStemmer, LancasterStemmer, RegexpStemmer, SnowballStemmer Study WordNetLemmatizer** |  |  |
| **9** | **Implement Naive Bayes classifier** |  |  |

**Practical No 1**

**Aim :**

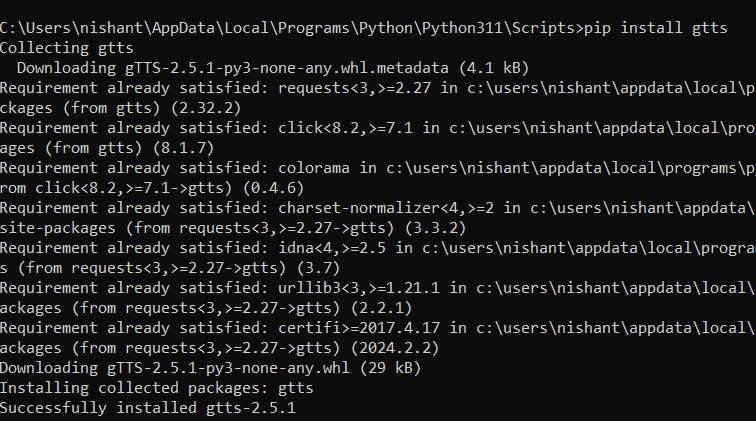
**a. Install NLTK**

**b. Convert the given text to speech**

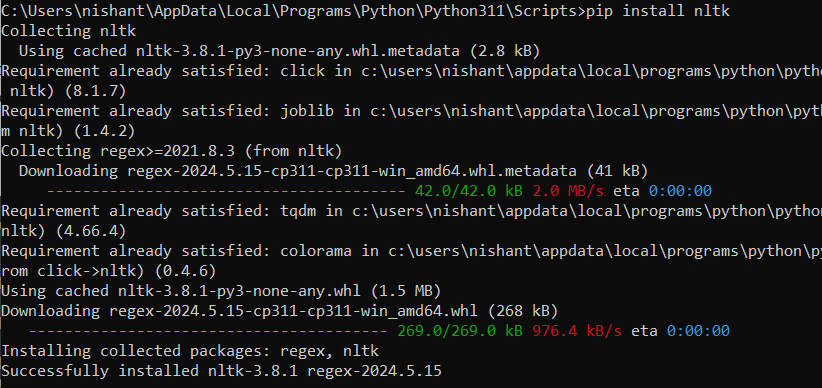
**c. Convert audio file Speech to Text.**

**a. Install NLTK**

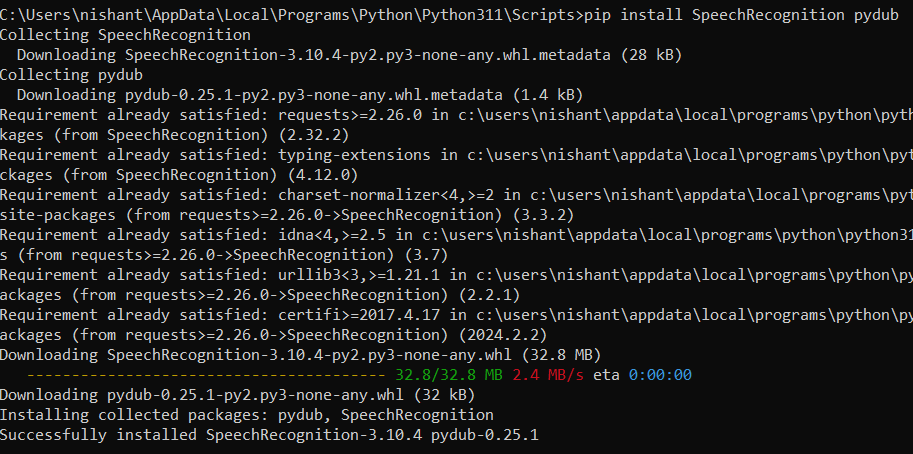
>pip install gtts



>pip install nltk



>pip install SpeechRecognition pydub



**b. Convert the given text to speech**

**Steps:**

**1. create file.txt in respective folder.**

**2. Enter some message in file.txt.**

**3. Save texttospeech.py file at same location.**

**Code :**

from gtts import gTTS

import os

f = open('1.txt')

x=f.read()

langauge='en'

audio=gTTS(text=x,lang=langauge)

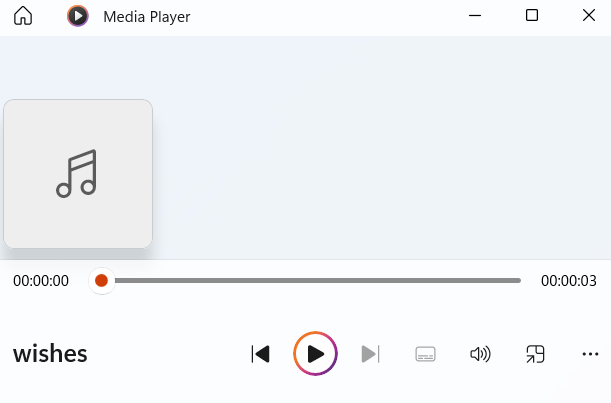
audio.save("wishes.wav")

os.system("wishes.wav")

print("program executed succesfully.")

**Output :**



****

**C. Convert audio file Speech to Text.**

**Code :**

import speech\_recognition as sr

filename = "Greetings.wav"

# initialize the recognizer

r = sr.Recognizer()

# open the file

with sr.AudioFile(filename) as source:

# listen for the data (load audio to memory)

audio\_data = r.record(source)

# recognize (convert from speech to text)

text = r.recognize\_google(audio\_data)

print(text)

Output :



**Practical No 2**

**a. Study of various Corpus – Brown, Inaugural, Reuters, udhr with various methods like**

**fields, raw, words, sents, categories.**

As just mentioned, a text corpus is a large body of text. Many corpora are designed to contain

a careful balance of material in one or more genres. We examined some small text collections, such as the speeches known as the US Presidential Inaugural Addresses.

This particular corpus actually contains dozens of individual texts — one per address — but for convenience we glued them end-to-end and treated them as a single text, also used various predefined texts that we accessed by typing from nltk.book import \*. However, since we want to be able to work with other texts, this section examines a variety of text corpora. We'll see how to select individual texts, and how to work with them.

The Brown Corpus is a balanced corpus consisting of 500 text samples from 15 different categories.

Fields: Metadata about the corpus.

Raw Text: Full raw text of a specific file.

Words: Tokenized words in the corpus.

Sentences: Tokenized sentences in the corpus.

Categories: Accessing different categories of the corpus.

**Code :**

import nltk

nltk.download('brown')

from nltk.corpus import brown

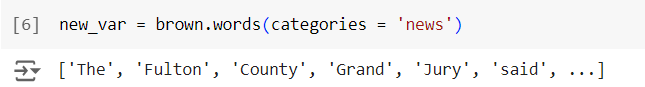
brown.categories()



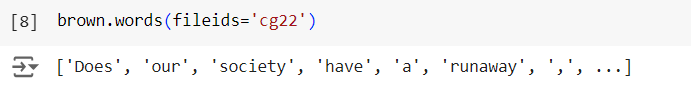
brown.fileids()



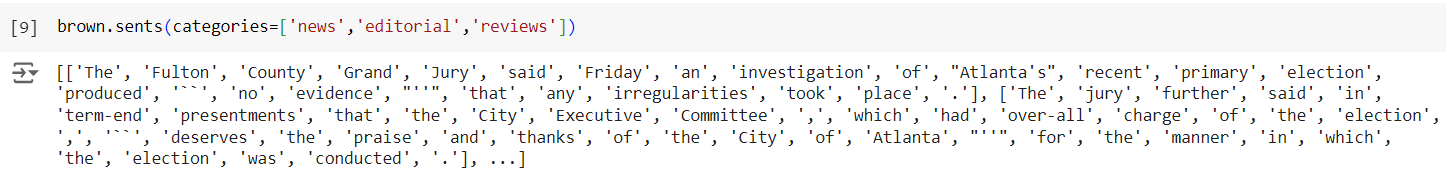
brown.words(categories = 'news')



brown.words(fileids='cg22')



brown.sents(categories=['news','editorial','reviews'])



**b. Create and use your own corpora(plaintext, categorical)**

Building a custom corpus can be useful for specific text analysis or natural language processing (NLP) tasks. In the following Python recipe, we are going to create a custom corpora which must be within one of the paths defined by NLTK. It is so because it can be found by NLTK. In order to avoid conflict with the official NLTK data package, let us create a custom natural\_language\_toolkit\_data directory in our home directory.

**Code :**

import os, os.path

path = os.path.expanduser('~/natural\_language\_toolkit\_data')

if not os.path.exists(path):

os.mkdir(path)

os.path.exists(path)

**Output :**



Now we will make a wordlist file, named wordfile.txt and put it in nltk\_data directory (content/wordfile.txt) and will load it by using nltk.data.load

**Corpus readers**

NLTK provides various CorpusReader classes.

**Creating wordlist corpus**

NLTK has WordListCorpusReader class that provides access to the file containing a list of

words. For the following we need to create a wordlist file which can be CSV or

normal text file. For example, we have created a file named ‘list’ that contains the following

data .

**Code :**

from nltk.corpus.reader import WordListCorpusReader

reader\_corpus = WordListCorpusReader('.',['wordfile.txt'])

reader\_corpus.words()

**Output :**



**c. Study Conditional frequency distributions**

Conditional frequency distributions (CFD) are a powerful tool in natural language processing and text analysis, allowing you to analyze the frequency of words or other features given specific conditions. The NLTK library in Python provides robust support for CFD through the nltk.probability.ConditionalFreqDist class.

**Code :**

#process a sequence of pairs

import nltk

nltk.download('inaugural')

nltk.download('udhr')

text = ['The', 'Fulton', 'County', 'Grand', 'Jury', 'said', ...]

pairs = [('news', 'The'), ('news', 'Fulton'), ('news', 'County'), ...]

import nltk

from nltk.corpus import brown

fd = nltk.ConditionalFreqDist(

(genre, word)

for genre in brown.categories()

for word in brown.words(categories=genre))

genre\_word = [(genre, word)

for genre in ['news', 'romance']

for word in brown.words(categories=genre)]

print(len(genre\_word))

print(genre\_word[:4])

print(genre\_word[-4:])

cfd = nltk.ConditionalFreqDist(genre\_word)

print(cfd)

print(cfd.conditions())

print(cfd['news'])

print(cfd['romance'])

print(list(cfd['romance']))

from nltk.corpus import inaugural

cfd = nltk.ConditionalFreqDist(

(target, fileid[:4])

for fileid in inaugural.fileids()

for w in inaugural.words(fileid)

for target in ['america', 'citizen']

if w.lower().startswith(target))

from nltk.corpus import udhr

languages = ['Chickasaw', 'English', 'German\_Deutsch',

'Greenlandic\_Inuktikut', 'Hungarian\_Magyar', 'Ibibio\_Efik']

cfd = nltk.ConditionalFreqDist(

(lang, len(word))

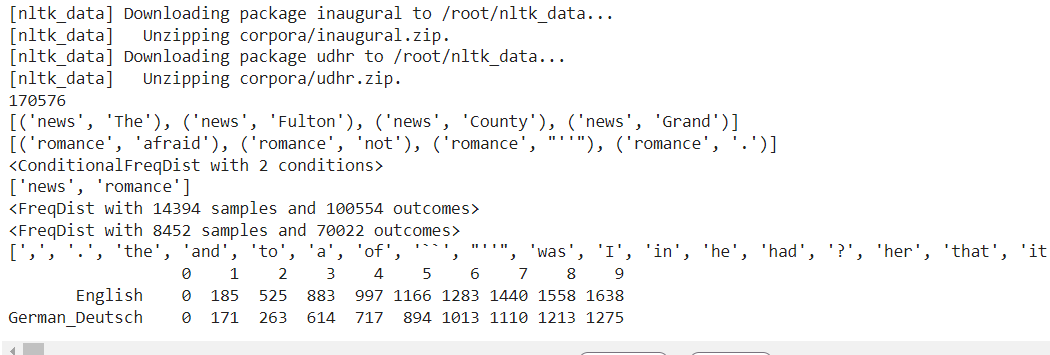
for lang in languages

for word in udhr.words(lang + '-Latin1'))

cfd.tabulate(conditions=['English', 'German\_Deutsch'],

samples=range(10), cumulative=True)

**Output :**



**Study of tagged corpora with methods like tagged\_sents, tagged\_words.**

Studying tagged corpora involves analyzing text data where each word is associated with a part-of-speech (POS) tag. Parts of speech are also known as word classes or lexical categories. The collection of tags used for a particular task is known as a tagset. This tagging provides valuable syntactic information that can be used for various natural language processing (NLP) tasks. NLTK offers several tagged corpora, such as the Brown Corpus, and provides methods like tagged\_sents and tagged\_words to access this tagged data.

**Tagged Words**: Returns a list of (word, tag) tuples.

**Tagged Sentences**: Returns a list of sentences, where each sentence is a list of (word, tag) tuples.

**Code :**

import nltk

from nltk import tokenize

nltk.download('punkt')

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

nltk.download('words')

para = "And now we are going to learn something new"

sents = tokenize.sent\_tokenize(para)

print("\nsentence tokenization\n===================\n",sents)

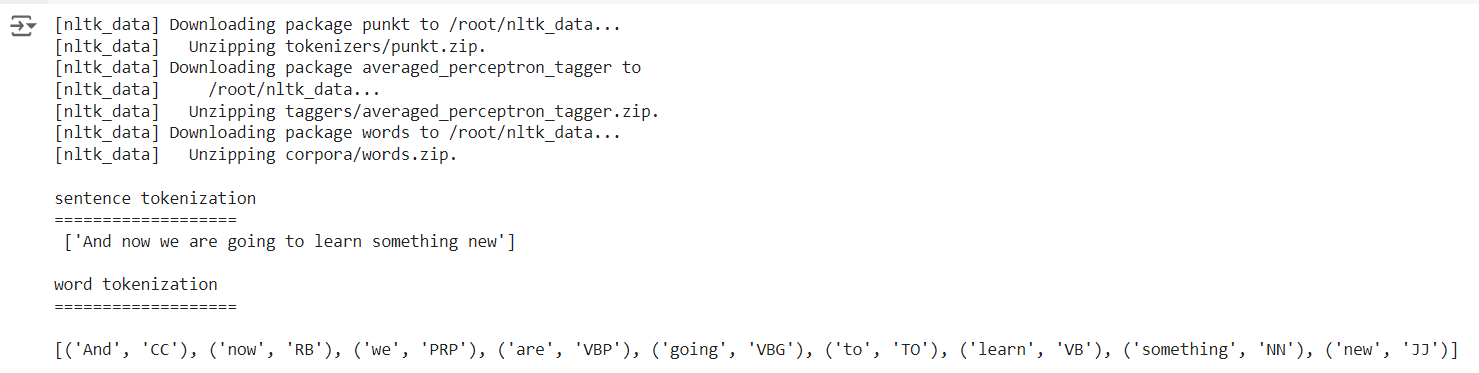
# word tokenization

print("\nword tokenization\n===================\n")

for index in range(len(sents)):

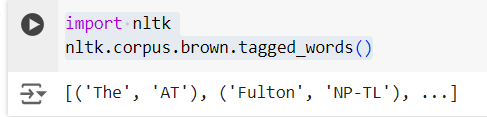
words = tokenize.word\_tokenize(sents[index])

print(nltk.pos\_tag(words))



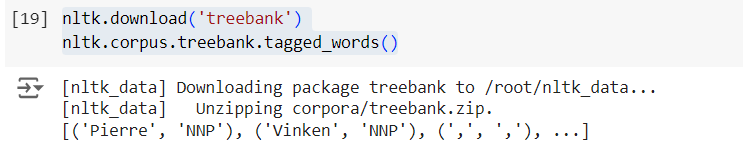
import nltk

nltk.corpus.brown.tagged\_words()



nltk.download('treebank')

nltk.corpus.treebank.tagged\_words()



**d. Write a program to find the most frequent noun tags.**

To find the most frequent noun tags in a tagged corpus using NLTK, you can follow these steps:

* Access the tagged words from a corpus.
* Filter out the noun tags.
* Compute the frequency distribution of these tags.
* Identify the most frequent noun tags.

**Code :**

nltk.download('universal\_tagset')

from nltk.corpus import brown

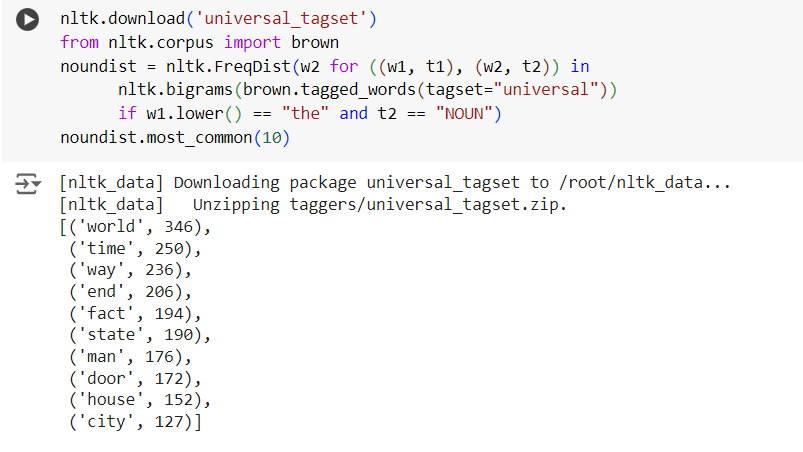
noundist = nltk.FreqDist(w2 for ((w1, t1), (w2, t2)) in

nltk.bigrams(brown.tagged\_words(tagset="universal"))

if w1.lower() == "the" and t2 == "NOUN")

noundist.most\_common(10)

**Output :**



**e. Map Words to Properties Using Python Dictionaries**

Mapping words to their properties using Python dictionaries is a common and powerful technique. You can create dictionaries where words are keys and their properties are values. These properties can include part-of-speech tags, frequencies, definitions, or any other relevant information.

**Code :**

#creating and printing a dictionary by mapping word with its properties

thisdict = {

"brand": "Ford",

"model": "Mustang",

"year": 1964

}

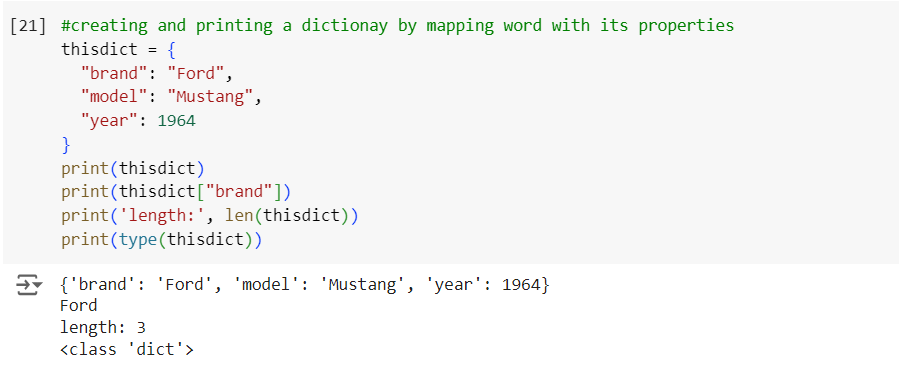
print(thisdict)

print(thisdict["brand"])

print('length:', len(thisdict))

print(type(thisdict))

**Output :**



**f. Study DefaultTagger, Regular expression tagger, UnigramTagger**

These taggers can be used individually or in combination to build more sophisticated taggers or to handle different tagging tasks effectively.

**A. DefaultTagger**

The DefaultTagger assigns a specific tag to every word in the text. It's useful as a baseline tagger or when dealing with unknown words.

**Code :**

from nltk.corpus import brown

nltk.download('brown')

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

nltk.FreqDist(tags).max()

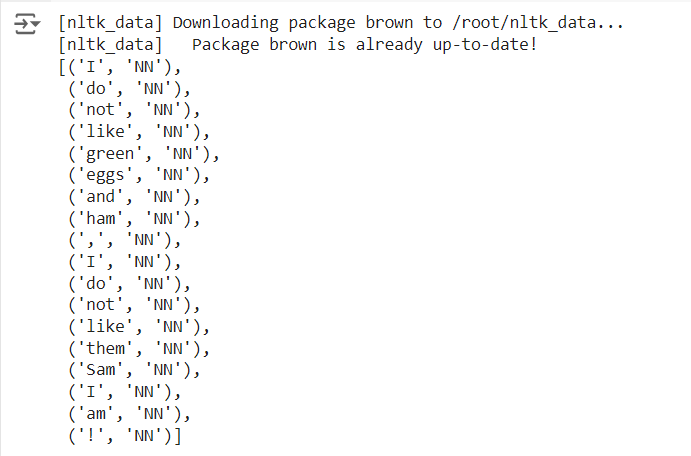
raw = 'I do not like green eggs and ham, I do not like them Sam I am!'

tokens = nltk.word\_tokenize(raw)

default\_tagger = nltk.DefaultTagger('NN')

default\_tagger.tag(tokens)

**Output :**



**B. Regular Expression Tagger:**

The Regular Expression Tagger assigns tags based on matching patterns defined using regular expressions. It's useful for tagging specific patterns of words.

**Code :**

import nltk

nltk.download('brown')

nltk.download('punkt')

from nltk.corpus import brown

from nltk import word\_tokenize

from nltk import RegexpTagger

brown\_sents = brown.sents(categories = 'news')

brown\_tagged\_sents = brown.tagged\_sents(categories = 'news')

import nltk

nltk.download('brown')

nltk.download('punkt')

from nltk.corpus import brown

from nltk import word\_tokenize

from nltk import RegexpTagger

brown\_sents = brown.sents(categories = 'news')

brown\_tagged\_sents = brown.tagged\_sents(categories = 'news')

patterns = [

(r'.\*ing$', 'VBG'), # gerunds

(r'.\*ed$', 'VBD'), # simple past

(r'.\*es$', 'VBZ'), # 3rd singular present

(r'.\*ould$', 'MD'), # modals

(r'.\*\'s$', 'NN$'), # possessive nouns

(r'.\*s$', 'NNS'), # plural nouns

(r'^-?[0-9]+(\.[0-9]+)?$', 'CD'), # cardinal numbers

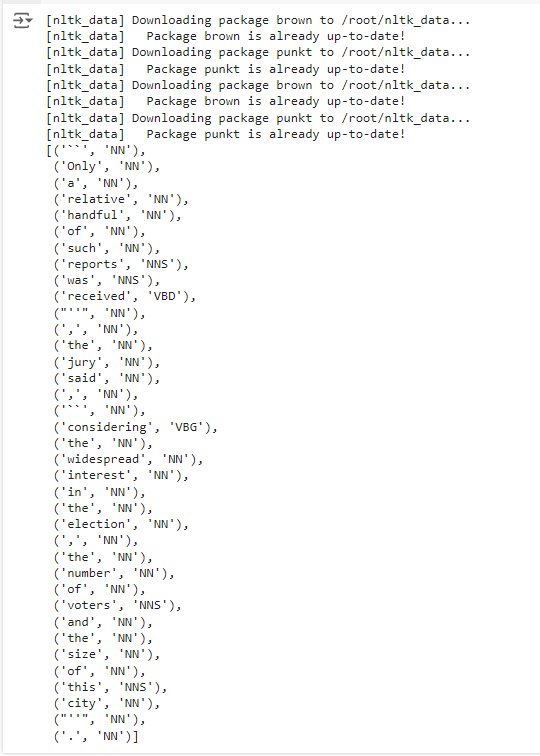
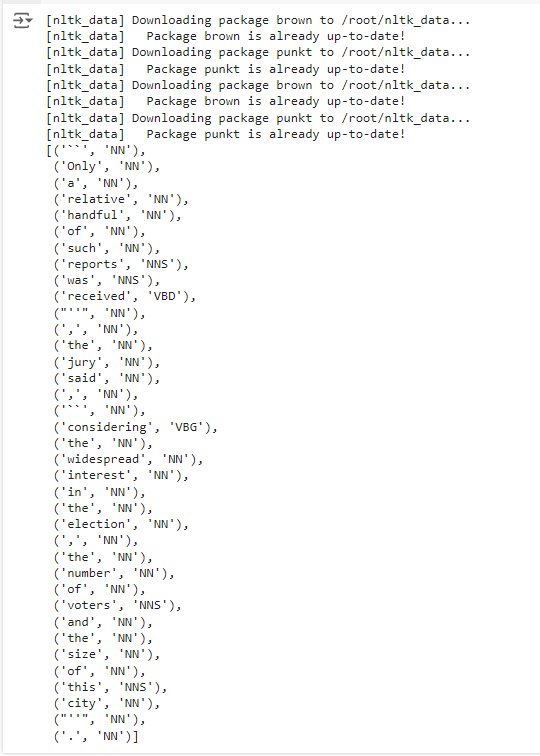
(r'.\*', 'NN') # nouns (default)

]

regexp\_tagger = nltk.RegexpTagger(patterns)

regexp\_tagger.tag(brown\_sents[3])

**Output :**



**C. Unigram Tagger**

The UnigramTagger assigns tags based on the most frequent tag for each word in the training data. It's a basic but effective tagger that often performs well, especially for languages with relatively free word order.

**Code :**

import nltk

nltk.download('brown')

nltk.download('punkt')

from nltk.corpus import brown

from nltk import UnigramTagger

brown\_tagged\_sents = brown.tagged\_sents(categories = 'news')

brown\_sents = brown.sents(categories = 'news')

unigram\_tagger = nltk.UnigramTagger(brown\_tagged\_sents)

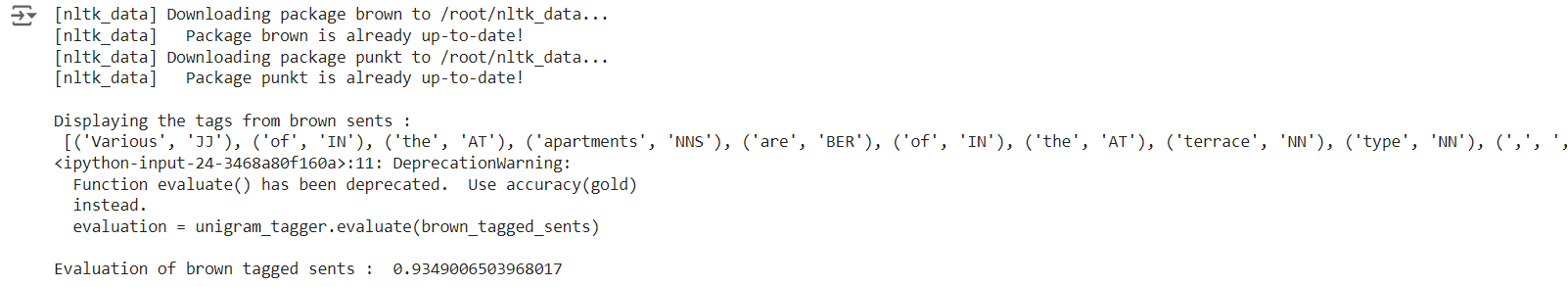
tags = unigram\_tagger.tag(brown\_sents[2007])

print("\nDisplaying the tags from brown sents : \n", tags)

evaluation = unigram\_tagger.evaluate(brown\_tagged\_sents)

print("\nEvaluation of brown tagged sents : ", evaluation)

**Output :**



**g. Find different words from a given plain text without any space by comparing this text with a given corpus of words. Also find the score of words.**

**Code :**

from \_\_future\_\_ import with\_statement #with statement for reading file

import re # Regular expression

words = [] # corpus file words

testword = [] # test words

ans = [] # words matches with corpus

print("MENU")

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

print(" 1 . Hash tag segmentation ")

print(" 2 . URL segmentation ")

print("enter the input choice for performing word segmentation")

choice = int(input())

if choice == 1:

text = "#whatismyname" # hash tag test data to segment

print("input with HashTag",text)

pattern=re.compile("[^\w']")

a = pattern.sub('', text)

elif choice == 2:

text = "www.whatismyname.com" # url test data to segment

print("input with URL",text)

a=re.split('\s|(?<!\d)[,.](?!\d)', text)

splitwords = ["www","com","in"] # remove the words which is containg in the list

a ="".join([each for each in a if each not in splitwords])

else:

print("wrong choice...try again")

print(a)

for each in a:

testword.append(each) #test word

test\_lenth = len(testword) # lenth of the test data

# Reading the corpus

with open('wordfile.txt', 'r') as f:

lines = f.readlines()

words =[(e.strip()) for e in lines]

def Seg(a,lenth):

ans =[]

for k in range(0,lenth+1): # this loop checks char by char in the corpus

if a[0:k] in words:

print(a[0:k],"-appears in the corpus")

ans.append(a[0:k])

break

if ans != []:

g = max(ans,key=len)

return g

test\_tot\_itr = 0 #each iteration value

answer = [] # Store the each word contains the corpus

Score = 0 # initial value for score

N = 37 # total no of corpus

M = 0

C = 0

while test\_tot\_itr < test\_lenth:

ans\_words = Seg(a,test\_lenth)

if ans\_words != 0:

test\_itr = len(ans\_words)

answer.append(ans\_words)

a = a[test\_itr:test\_lenth]

test\_tot\_itr += test\_itr

Aft\_Seg = " ".join([each for each in answer])

# print segmented words in the list

print("output")

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

print(Aft\_Seg) # print After segmentation the input

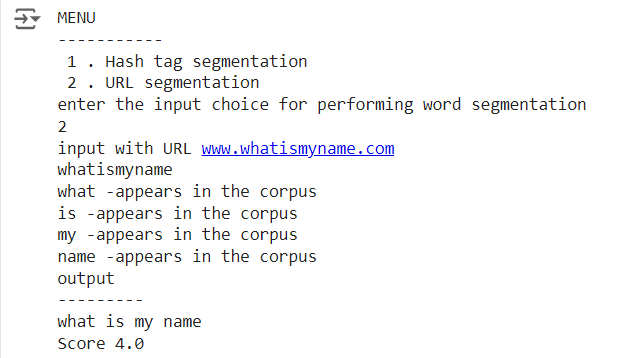
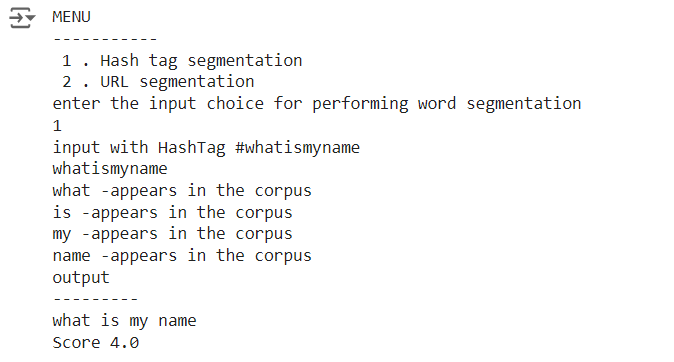
# Calculating Score

C = len(answer)

score = C \* N / N # Calculate the score

print("Score",score)

**Output :**



**Practical No 3**

**Aim:**

**a. Study of Wordnet Dictionary with methods as synsets, definitions, examples, antonyms.**

**b. Study lemmas, hyponyms, hypernyms, entailments,**

**c. Write a program using python to find synonym and antonym of word "active" using Wordnet**

**d. Compare two nouns**

**e. Handling stopword.**

**Using nltk Adding or Removing Stop Words in NLTK's Default Stop Word List**

**Using Gensim Adding and Removing Stop Words in Default Gensim Stop Words List**

**Using Spacy Adding and Removing Stop Words in Default Spacy Stop Words List**

**a. Study of Wordnet Dictionary with methods as synsets, definitions, examples, antonyms.**

WordNet is a large lexical database of English, developed at Princeton University. It groups English words into sets of synonyms called synsets, provides short definitions and usage examples, and records a number of relations among these synonym sets or their members.

**Synsets** : Synsets are sets of cognitive synonyms (synonyms that express the same concept). Each synset represents a distinct concept and contains a group of words that are interchangeable in some context.

**Definitions** : Each synset comes with a short definition that explains the concept it represents.

**Examples** : WordNet provides usage examples for synsets to illustrate how the words in the synset can be used in context.

**Antonyms** : WordNet includes antonyms for some words. Antonyms are words with opposite meanings.

**Code :**

import nltk

nltk.download('wordnet')

from nltk.corpus import wordnet

print(wordnet.synsets("computer"))

# definition and example of the word ‘computer’

print(wordnet.synset("computer.n.01").definition())

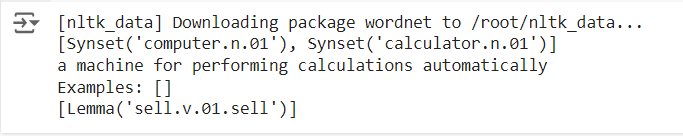
#examples

print("Examples:", wordnet.synset("computer.n.01").examples())

#get Antonyms

print(wordnet.lemma('buy.v.01.buy').antonyms())

**Output :**



**b. Study lemmas, hyponyms, hypernyms, entailments**

Lemmatization usually refers to doing things properly with the use of a vocabulary and

morphological analysis of words, normally aiming to remove inflectional endings only and to

return the base or dictionary form of a word, which is known as the lemma .

Hyponyms: More specific terms. Both come to picture as Synsets are organized in a structure

similar to that of an inheritance tree. This tree can be traced all the way up to a root hypernym.

Hypernyms provide a way to categorize and group words based on their similarity to each

other.

Hypernym extraction is a crucial task for semantically motivated NLP tasks such as taxonomy

and ontology learning, textual entailment or paraphrase identification. ... Our experiments

confirm that both syntactic and definitional information play a crucial role in

the hypernym extraction task.

Textual entailment (TE) in natural language processing is a directional relation between text

fragments. The relation holds whenever the truth of one text fragment follows from another

text. In the TE framework, the entailing and entailed texts are termed text (t) and hypothesis

(h), respectively.

**Code :**

import nltk

from nltk.corpus import wordnet

print(wordnet.synsets("computer"))

print(wordnet.synset("computer.n.01").lemma\_names())

#all lemmas for each synset.

for e in wordnet.synsets("computer"):

print(f'{e} --> {e.lemma\_names()}')

#print all lemmas for a given synset

print(wordnet.synset('computer.n.01').lemmas())

#get the synset corresponding to lemma

print(wordnet.lemma('computer.n.01.computing\_device').synset())

#Get the name of the lemma

print(wordnet.lemma('computer.n.01.computing\_device').name())

#Hyponyms give abstract concepts of the word that are much more specific

#the list of hyponyms words of the computer

syn = wordnet.synset('computer.n.01')

print(syn.hyponyms)

print([lemma.name() for synset in syn.hyponyms() for lemma in synset.lemmas()])

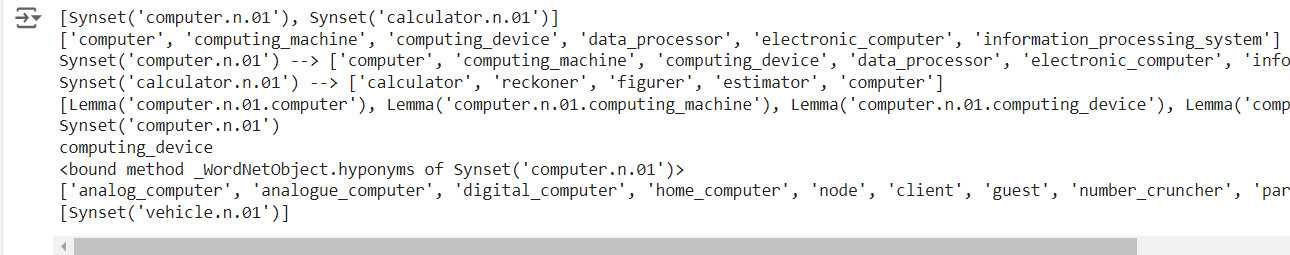
#the semantic similarity in WordNet

vehicle = wordnet.synset('vehicle.n.01')

car = wordnet.synset('car.n.01')

print(car.lowest\_common\_hypernyms(vehicle))

**Output :**



**C. Write a program using python to find synonym and antonym of word "active" using Wordnet**

To find synonyms and antonyms of the word "active" using WordNet in Python, we can utilize the Natural Language Toolkit (nltk) library, which provides a simple interface to the WordNet lexical database.

**Code :**

from nltk.corpus import wordnet

# Retrieve synsets for the word "active"

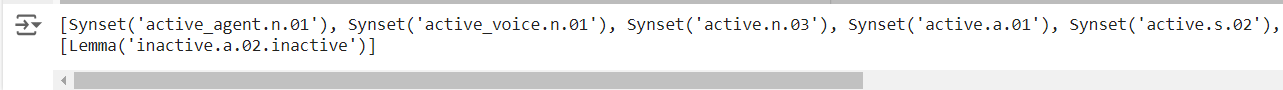
# retrieves all the synsets (sets of cognitive synonyms) for the word "active". A synset contains a group of synonyms that share a common meaning. The result is a list of synsets.

print( wordnet.synsets("active"))

# Find antonyms for a specific lemma

print(wordnet.lemma('active.a.01.active').antonyms()) #This retrieves a specific lemma (a word form with a specific sense) from the synset active.a.01. Here, 'active' is the lemma name, 'a' stands for adjective, and '01' is the sense number.

**Output :**



**d. Compare two nouns**

**Code :**

import nltk

nltk.download('wordnet')

from nltk.corpus import wordnet

# Get synsets for 'football' and 'soccer'

syn1 = wordnet.synsets('football')

syn2 = wordnet.synsets('soccer')

# A word may have multiple synsets, so need to compare each synset of word1 with synset of word2

for s1 in syn1:

for s2 in syn2:

print("Path similarity of: ")

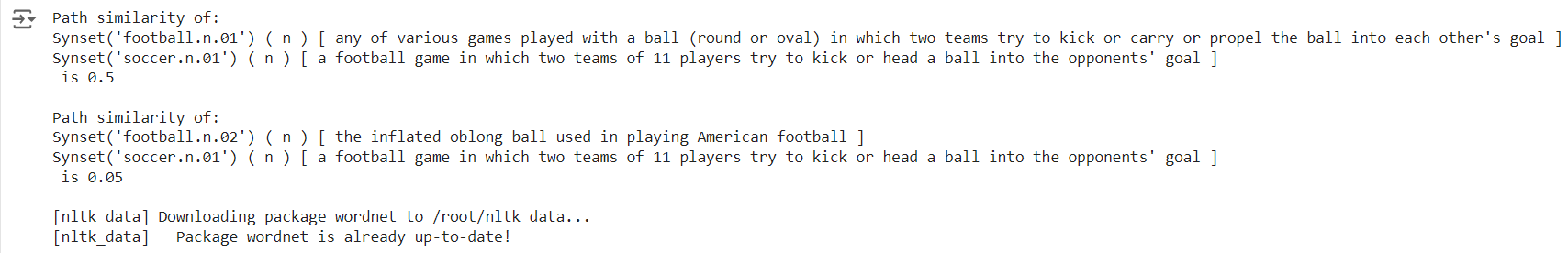
print(s1, '(', s1.pos(), ')', '[', s1.definition(), ']') #Each synset has a name (e.g., Synset('football.n.01')), part of speech (e.g., noun 'n'), and a definition.

print(s2, '(', s2.pos(), ')', '[', s2.definition(), ']')

print(" is", s1.path\_similarity(s2))

print()

**Output :**



**e. Handling stopword.**

**Using nltk Adding or Removing Stop Words in NLTK's Default Stop Word List**

NLTK provides a default set of stop words for English, which can be useful for various natural language processing tasks like text classification, sentiment analysis, and information retrieval. These stop words are common words that are often removed from text because they typically do not carry significant meaning.

**Code :**

import nltk

nltk.download('punkt')

from nltk.corpus import stopwords

nltk.download('stopwords')

from nltk.tokenize import word\_tokenize

text = "Yashesh likes to play football, however he is not too fond of tennis."

text\_tokens = word\_tokenize(text)

tokens\_without\_sw = [word for word in text\_tokens if not word in

stopwords.words()]

print(tokens\_without\_sw)

#add the word 'play' to the NLTK stop word collection

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

all\_stopwords.append('play')

text\_tokens = word\_tokenize(text)

tokens\_without\_sw = [word for word in text\_tokens if not word in all\_stopwords]

print(tokens\_without\_sw)

#remove ‘not’ from stop word collection

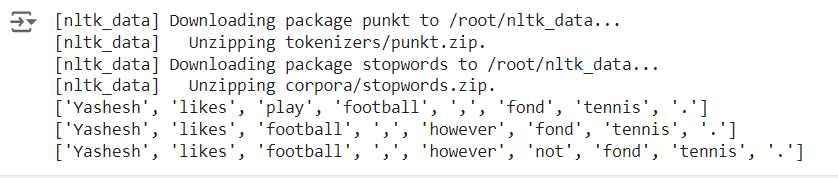
all\_stopwords.remove('not')

text\_tokens = word\_tokenize(text)

tokens\_without\_sw = [word for word in text\_tokens if not word in all\_stopwords]

print(tokens\_without\_sw)

**Output :**



**Using Gensim Adding and Removing Stop Words in Default Gensim Stop Words List**

Gensim, a popular library for topic modeling and natural language processing tasks, also provides a set of default stop words. Similar to NLTK, you can add or remove words from Gensim's default stop words list to customize it according to your needs.

**Code :**

import gensim

from gensim.parsing.preprocessing import remove\_stopwords

text = "Yashesh likes to play football, however he is not too fond of tennis."

print('Original text: ', text)

filtered\_sentence = remove\_stopwords(text)

print('\n After removing Stop words: ',filtered\_sentence)

#The below line retrieves the default set of stop words from Gensim and prints them.

all\_stopwords = gensim.parsing.preprocessing.STOPWORDS

print('\n Stop words in Gensim: ', all\_stopwords)

#'''The following script adds likes and play to the list of stop words in Gensim:'''

from gensim.parsing.preprocessing import STOPWORDS

all\_stopwords\_gensim = STOPWORDS.union(set(['likes', 'play'])) # adding 'likes' and 'play' to stop words

text = "Yashesh likes to play football, however he is not too fond of tennis."

text\_tokens = word\_tokenize(text)

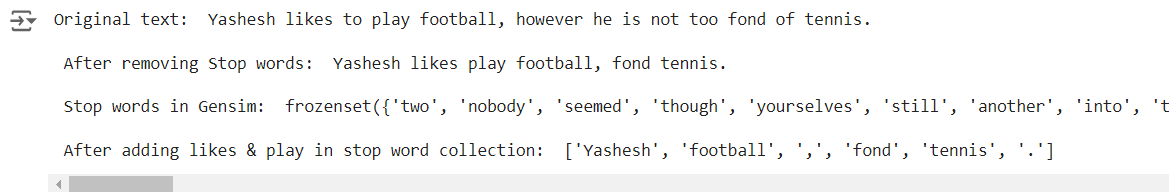
# Filter out the tokens that are in the new stop words set (including 'likes' and 'play')

tokens\_without\_sw = [word for word in text\_tokens if not word in

all\_stopwords\_gensim]

print("\n After adding likes & play in stop word collection: ",tokens\_without\_sw)

**Output :**



# Remove the word 'not' from the existing set of Gensim stop words

from gensim.parsing.preprocessing import STOPWORDS

all\_stopwords\_gensim = STOPWORDS

sw\_list = {"not"}

all\_stopwords\_gensim = STOPWORDS.difference(sw\_list)

text = "Yashesh likes to play football, however he is not too fond of tennis."

text\_tokens = word\_tokenize(text)

# Filter out the tokens again with 'not' removed from stop words set

tokens\_without\_sw = [word for word in text\_tokens if not word in

all\_stopwords\_gensim]

print(tokens\_without\_sw)



**Using Spacy Adding and Removing Stop Words in Default Spacy Stop Words List**

Stop words are commonly used words in a language, such as "the," "is," "and," or "in," which do not contribute much to the overall meaning of the text. Stop word removal can improve the accuracy and efficiency of text analysis, text mining, and natural language processing tasks.

**Code :**

#python -m spacy download en\_core\_web\_sm

import spacy

import nltk

from nltk.tokenize import word\_tokenize

sp = spacy.load('en\_core\_web\_sm')

#add the word play to the NLTK stop word collection

all\_stopwords = sp.Defaults.stop\_words

all\_stopwords.add("play")

text = "Yashesh likes to play football, however he is not too fond of tennis."

text\_tokens = word\_tokenize(text)

tokens\_without\_sw = [word for word in text\_tokens if not word in all\_stopwords]

print(tokens\_without\_sw)

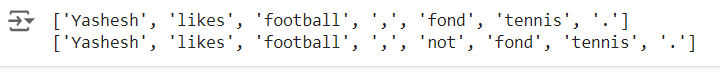
#remove 'not' from stop word collection

all\_stopwords.remove('not')

tokens\_without\_sw = [word for word in text\_tokens if not word in all\_stopwords]

print(tokens\_without\_sw)

**Output :**



**Practical 04**

**Aim: Text Tokenization**

**a. Tokenization using Python’s split() function**

Tokenization is the process of splitting text into smaller units called tokens, which can be words, phrases, or even characters. One of the simplest methods to perform tokenization in Python is by using the split() function. This function divides a string into a list of substrings based on a specified delimiter. By default, split() uses whitespace as the delimiter.

**Code:**

text = """Founded in 2002, SpaceX’s mission is to enable humans to become a spacefaring civilization and a multi-planet species by building a self-sustaining city on Mars. In 2008, SpaceX’s Falcon 1 became the first privately developed liquid-fuel launch vehicle to orbit the Earth."""

# Splits at space

a=text.split()

print(a)

**Output :**



**b. Tokenization using Regular Expressions (RegEx)**

Tokenization using Regular Expressions (RegEx) is a powerful and flexible method for splitting text into tokens based on complex patterns. Regular expressions allow you to define patterns for identifying the tokens you want to extract from a string. This method is particularly useful for handling a variety of delimiters, removing unwanted characters, and performing more complex text processing tasks.

**Code:**

import re

text = """Founded in 2002, SpaceX’s mission is to enable humans to become a spacefaring civilization and a multiplanet

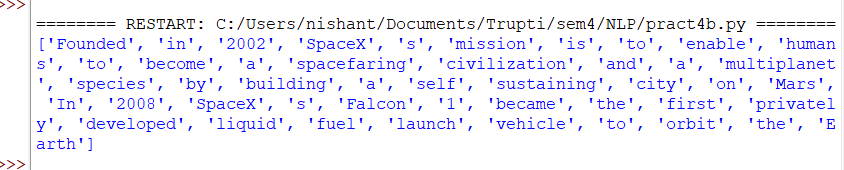
species by building a self-sustaining city on Mars. In 2008, SpaceX’s Falcon 1 became the first privately developed

liquid-fuel launch vehicle to orbit the Earth."""

tokens = re.findall("[\w']+", text)

print(tokens)

**Output :**



**c. Tokenization using NLTK**

The Natural Language Toolkit (NLTK) is a comprehensive library for natural language processing in Python. It provides powerful tools for text processing, including tokenization, which is the process of splitting text into individual tokens (words, sentences, etc.). NLTK offers several built-in tokenizers that handle various aspects of tokenization.

**Code :**

import nltk

nltk.download('punkt')

from nltk.tokenize import word\_tokenize

text = """Founded in 2002, SpaceX’s mission is to enable humans to become a spacefaring civilization and a multi-

planet

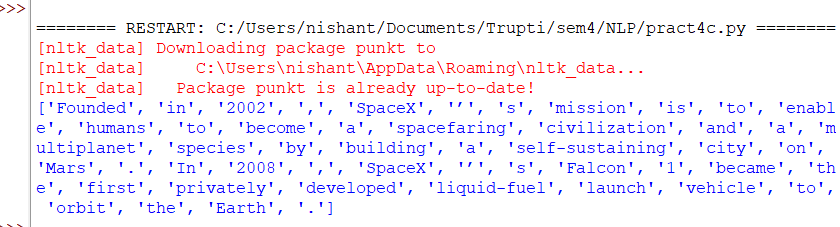
species by building a self-sustaining city on Mars. In 2008, SpaceX’s Falcon 1 became the first privately developed

liquid-fuel launch vehicle to orbit the Earth."""

a=word\_tokenize(text)

print(a)

**Output :**



**d. Tokenization using the spaCy library**

spaCy is a powerful and efficient library for natural language processing in Python. It provides a variety of advanced NLP capabilities, including tokenization, which is the process of breaking down text into individual tokens (such as words and punctuation marks). spaCy's tokenization is particularly robust, as it handles a wide range of linguistic features out-of-the-box.

**Code :**

#pip install -U spacy

#python -m spacy download en

from spacy.lang.en import English

# Load English tokenizer, tagger, parser, NER and word vectors

nlp = English()

text = """Founded in 2002, SpaceX’s mission is to enable humans to become a spacefaring civilization and a multi-planet

species by building a self-sustaining city on Mars. In 2008, SpaceX’s Falcon 1 became the first privately developed

liquid-fuel launch vehicle to orbit the Earth."""

# "nlp" Object is used to create documents with linguistic annotations.

my\_doc = nlp(text)

# Create list of word tokens

token\_list = []

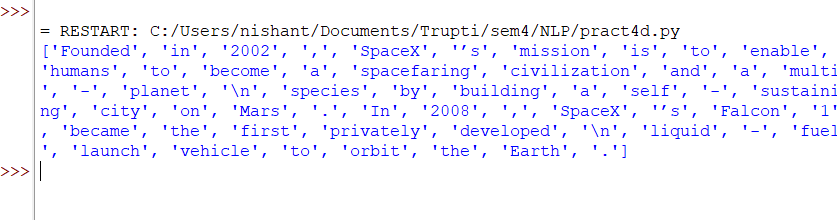
for token in my\_doc:

token\_list.append(token.text)

token\_list

print(token\_list)

**Output :**



**e. Tokenization using Keras.**

Keras, a popular deep learning library, includes utilities for text preprocessing, including tokenization. The keras.preprocessing.text module provides the Tokenizer class, which can convert text into sequences of tokens suitable for training machine learning models.

**Code :**

from keras.preprocessing.text import text\_to\_word\_sequence

# define

text = """Founded in 2002, SpaceX’s mission is to enable humans to become a spacefa

ring civilization and a multi-planet

species by building a selfsustaining city on Mars. In 2008, SpaceX’s Falcon 1 became the first privately deve

loped

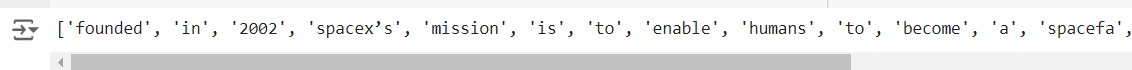
liquid-fuel launch vehicle to orbit the Earth."""

# tokenize

result = text\_to\_word\_sequence(text)

print(result)

**Output :**



**f. Tokenization using Gensim.**

Gensim is a robust library for topic modeling and document similarity analysis in Python, and it includes utilities for text preprocessing, including tokenization. Gensim's gensim.utils module provides simple and effective functions for tokenizing text.

**Code :**

from gensim.utils import tokenize

text = """Founded in 2002, SpaceX’s mission is to enable humans to become a spacefa

ring civilization and a multi-planet

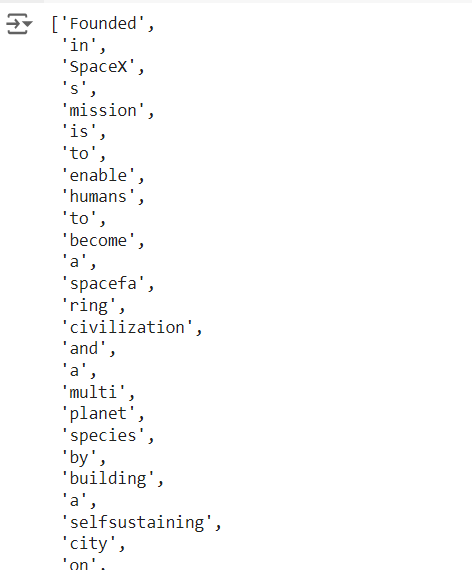
species by building a selfsustaining city on Mars. In 2008, SpaceX’s Falcon 1 became the first privately deve

loped

liquid-fuel launch vehicle to orbit the Earth."""

list(tokenize(text))

**Output :**



**Practical No 5**

**Aim: Illustrate part of speech tagging.**

a. Part of speech Tagging and chunking of user defined text.

b. Named Enltity recognition of user defined text.

c. Named Entity recognition with diagram using NLTK corpus – treebank

**a. Part of speech Tagging and chunking of user defined text.**

Theory: POS Tagging (Parts of Speech Tagging) is a process to mark up the words in text format for a particular part of a speech based on its definition and context. It is responsible for text reading in a language and assigning some specific token (Parts of Speech) to each word. It is also called grammatical tagging.

**Code :**

import nltk

from nltk import pos\_tag

from nltk import RegexpParser

nltk.download()

text ="This is practical no 6".split()

print("After Split:",text)

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

# averaged\_perceptron\_tagger is used for tagging words with their parts of speech (POS)

tokens\_tag = pos\_tag(text)

print("After Token:",tokens\_tag)

patterns= """mychunk:{<NN.?>\*<VBD.?>\*<JJ.?>\*<CC>?}"""

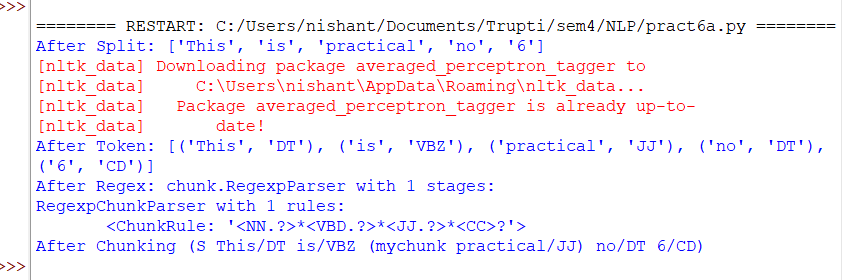
chunker = RegexpParser(patterns)

print("After Regex:",chunker)

output = chunker.parse(tokens\_tag)

print("After Chunking",output)

**Output:**



**b. Named Enltity recognition of user defined text.**

Theory: Named-entity recognition is a subtask of information extraction that seeks to locate and classify named entities mentioned in unstructured text into pre-defined categories such as person names, organizations, locations, medical codes, time expressions, quantities, monetary values, percentages, etc. European is NORD (nationalities or religious or political groups), Google is an organization, $5.1 billion is monetary value and Wednesday is a date object. They are all correct.

**Code :**

import nltk

import spacy

from spacy import displacy

from collections import Counter

import en\_core\_web\_sm

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

ex = 'European authorities fined Google a record $5.1 billion on Wednesday for abusing its power in the mobile phone market and ordered the company to alter its practices'

ex = nlp('European authorities fined Google a record $5.1 billion on Wednesday for abusingits power in the mobile phone market and ordered the company to alter its practices')

print([(X.text, X.label\_) for X in ex.ents])

**Output :**



**c. Named Entity recognition with diagram using NLTK corpus – treebank**

Theory: The maxent\_ne\_chunker contains two pre-trained English named entity chunkers trained on an ACE corpus.

**Code :**

sentence = 'Peterson first suggested the name "open source" at Palo Alto, California'

import nltk

nltk.download('punkt')

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

words = nltk.word\_tokenize(sentence)

pos\_tagged = nltk.pos\_tag(words)

nltk.download('maxent\_ne\_chunker')

nltk.download('words')

ne\_tagged = nltk.ne\_chunk(pos\_tagged)

print("NE tagged text:")

print(ne\_tagged)

print()

print("Recognized named entities:")

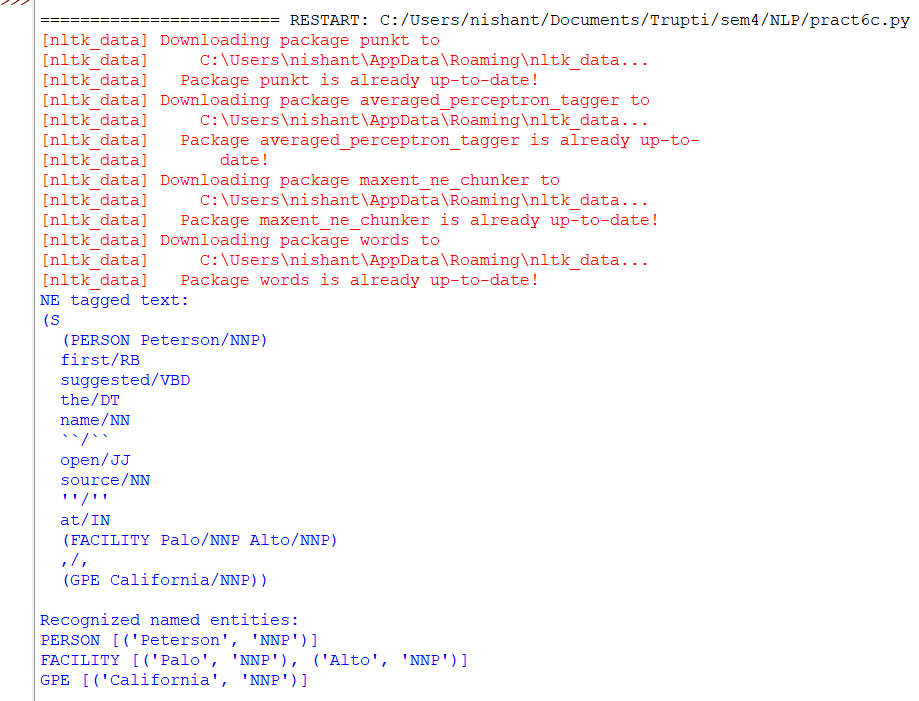
for ne in ne\_tagged:

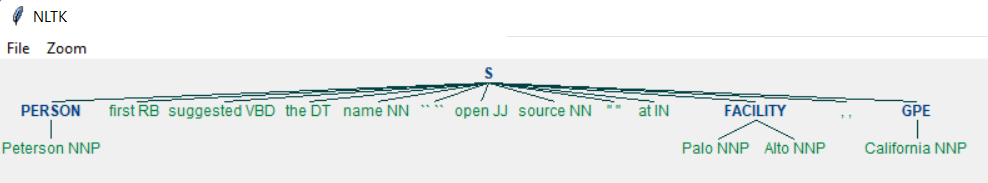
if hasattr(ne, "label"):

print(ne.label(), ne[0:])

ne\_tagged.draw()

**Output :**





**Practical No 6**

**Aim:**

**a. Define grammer using nltk. Analyze a sentence using the same.**

**b. Accept the input string with Regular expression of FA: 101+**

**c. Accept the input string with Regular expression of FA: (a+b)\*bba**

**d. Implementation of Deductive Chart Parsing using context free grammar and a given sentence.**

**a. Define grammer using nltk. Analyze a sentence using the same.**

To define grammar using NLTK (Natural Language Toolkit), we typically create a context-free grammar (CFG) using a syntax called Backus-Naur Form (BNF). Here's a simple example of defining a CFG for a basic English sentence:

**Code :**

# using Recursive Descent Parser

import nltk

grammar1 = nltk.CFG.fromstring("""

S -> NP VP

VP -> V NP | V NP PP

PP -> P NP

V -> "saw" | "ate" | "walked"

NP -> "John" | "Mary" | "Bob" | Det N | Det N PP

Det -> "a" | "an" | "the" | "my"

N -> "man" | "dog" | "cat" | "telescope" | "park"

P -> "in" | "on" | "by" | "with"

""")

sent = "Mary saw Bob".split()

rd\_parser = nltk.RecursiveDescentParser(grammar1)

for tree in rd\_parser.parse(sent):

print(tree)



import nltk

nltk.download('punkt')

from nltk import tokenize

grammar1 = nltk.CFG.fromstring("""

S -> VP

VP -> VP NP

NP -> Det NP

Det -> 'that'

NP -> singular Noun

NP -> 'flight'

VP -> 'Book'

""")

sentence = "Book that flight"

for index in range(len(sentence)):

all\_tokens = tokenize.word\_tokenize(sentence)

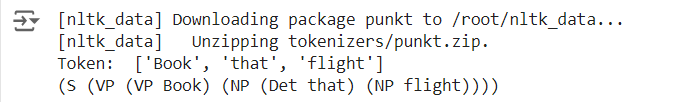
print("Token: ",all\_tokens)

parser = nltk.ChartParser(grammar1)

for tree in parser.parse(all\_tokens):

print(tree)

#tree.draw()



**b. Accept the input string with Regular expression of FA: 101+**

allows you to accept input strings that follow the regular expression pattern 101+

**Code :**

def FA(s):

# if the length is less than 3 then it can't be accepted, Therefore end the process.

if len(s) < 3:

return "Rejected"

# first three characters are fixed. Therefore checking them using index

if s[0] == '1': # Check if the first character is '1'

if s[1] == '0': # Check if the second character is '0'

if s[2] == '1': # Check if the third character is '1'

# After index 2 only "1" can appear. Therefore break the process if any other character is detected

for i in range(3, len(s)): # Loop through the remaining characters

if s[i] != '1': # If any character is not '1'

return "Rejected" # Reject the string

return "Accepted" # If all characters after the first three are '1', accept the string

# If any of the conditions fail, return "Rejected"

return "Rejected"

inputs=['1','10101','101','10111','01010','100','','10111101','1011111']

# Loop through each input string and print whether it is accepted or rejected

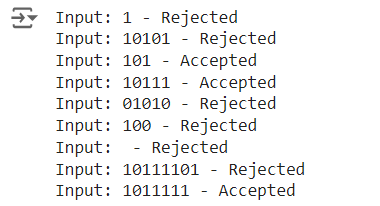
#for i in inputs:

# print(FA(i))

for i in inputs:

print(f"Input: {i} - {FA(i)}")

**Output :**



**c. Accept the input string with Regular expression of FA: (a+b)\*bba.**

code allows you to accept input strings that follow the regular expression pattern (a+b)\*bba.

The regular expression (a+b)\*bba can be broken down as follows:

(a+b)\*: Zero or more occurrences of either 'a' or 'b'.

bba: The string must end with the sequence 'bba'.

**Code :**

def FA(s):

size = 0

# Scan the complete string and make sure that it contains only 'a' & 'b'

for i in s:

if i == 'a' or i == 'b':

size += 1

else:

return "Rejected"

# After checking that it contains only 'a' & 'b'

# Check its length, it should be at least 3

if size >= 3:

# Check the last 3 elements

if s[size-3] == 'b':

if s[size-2] == 'b':

if s[size-1] == 'a':

return "Accepted" # If all 4 conditions are true

return "Rejected" # Else of 4th if

return "Rejected" # Else of 3rd if

return "Rejected" # Else of 2nd if

return "Rejected" # Else of 1st if

# List of input strings to test

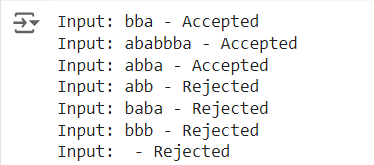
inputs = ['bba', 'ababbba', 'abba', 'abb', 'baba', 'bbb', '']

# Iterate over the input strings and print the result of FA for each

for i in inputs:

print(f"Input: {i} - {FA(i)}")

**Output :**



**d. Implementation of Deductive Chart Parsing using context free grammar and a given sentence.**

Deductive Chart Parsing is a method for parsing sentences using a chart data structure, which stores partial parse trees as they are constructed.

**Code :**

import nltk

from nltk import CFG, ChartParser, word\_tokenize

# Define the context-free grammar

grammar1 = CFG.fromstring("""

S -> NP VP

PP -> P NP

NP -> Det N | Det N PP | 'I'

VP -> V NP | VP PP

Det -> 'a' | 'my'

N -> 'bird' | 'balcony'

V -> 'saw'

P -> 'in'

""")

# Define the sentence and tokenize it

sentence = "I saw a bird in my balcony"

all\_tokens = word\_tokenize(sentence)

print(all\_tokens)

# Initialize the chart parser with the grammar

parser = ChartParser(grammar1)

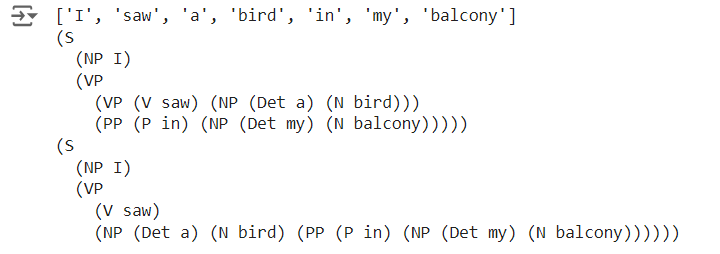
# Parse the tokens and print the parse trees

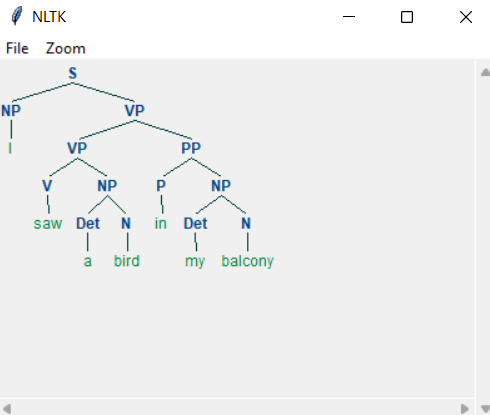
for tree in parser.parse(all\_tokens):

print(tree)

#tree.draw()

**Output :**





**Practical No 7**

**PART A : STUDY PORTER STEMMER, LANCASTER STEMMER, REGEXP STEMMER AND SNOWBALL STEMMER**

**a. Porter Stemmer**

The Porter Stemmer, developed by Martin Porter in 1980, is a widely used algorithm in natural language processing (NLP) for reducing words to their root forms, or stems. This process helps in standardizing words and improving the performance of text-based applications, such as search engines and information retrieval systems.

**Code :**

# Program to implement Porter Stemmer

print("Name : Trupti Bhostekar \tRoll No : 2")

import nltk

nltk.download('punkt')

from nltk.stem import PorterStemmer

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

# Defining the stemmer

porter = PorterStemmer()

# Taking words which have a similar stem

terms = ["gene", "genes", "genesis", "genetic", "generic", "general"]

# Performing stemming using porter stemmer on words

print("\n1. Performing porter stemming on the words")

for each\_term in terms:

print(porter.stem(each\_term))

# Taking a sentence

sentence = "Heya Reeba, do you know it is important to be pythonly while pythoning with pythonlanguage. Stay being a pythoner"

# Performing stemming using porter stemmer on a sentence

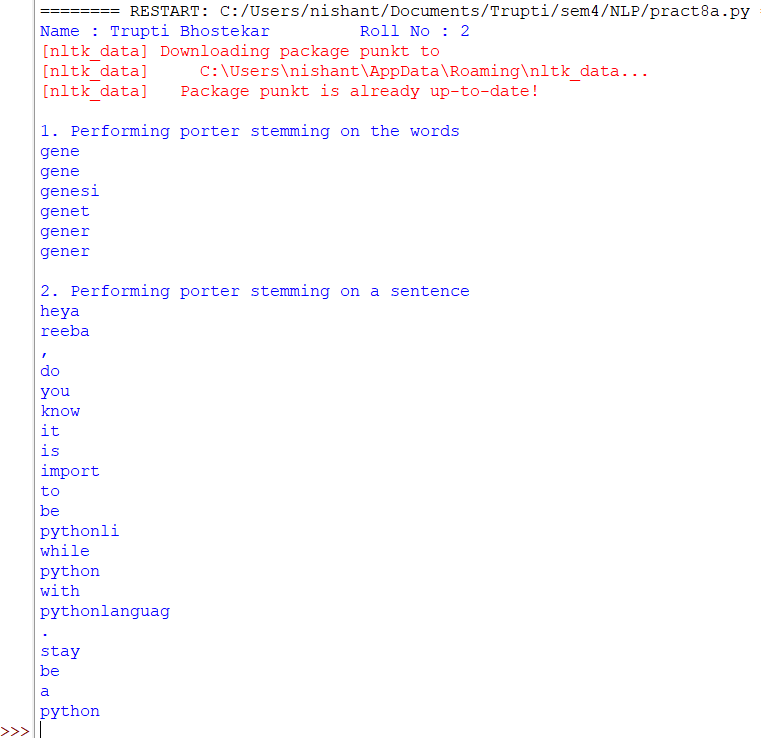
print("\n2. Performing porter stemming on a sentence")

words = word\_tokenize(sentence, language = 'english')

for each\_word in words:

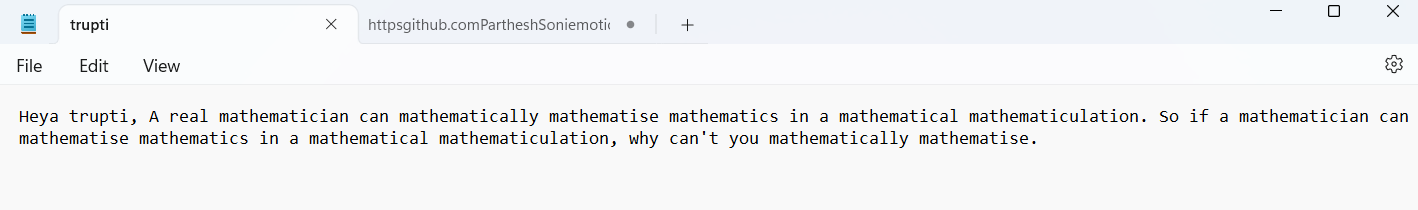
print(porter.stem(each\_word))

**Output :**



**b. Lancaster Stemmer**

The Lancaster Stemmer, also known as the Paice-Husk Stemmer, is another widely used stemming algorithm in natural language processing (NLP). It was developed by Chris Paice and Gareth Husk in 1990 and is known for its aggressive stemming approach, which often results in shorter stems compared to other algorithms like the Porter Stemmer.



**Code :**

# Program to implement Lancaster Stemmer

print("Name : Trupti Bhostekar \tRoll No : 2")

import nltk

nltk.download('punkt')

from nltk.stem import LancasterStemmer

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

# Defining the stemmer

lancaster = LancasterStemmer()

# Taking words which have a similar stem

terms = ["enjoy", "enjoying", "enjoyed", "enjoyable", "enjoyment", "enjoyful"]

# Performing stemming using lancaster stemmer on words

print("\n1. Performing lancaster stemming on the words")

for each\_term in terms:

print(lancaster.stem(each\_term))

# Taking a sentence

sentence = "Heya trupti, Why is it so with the dancers that when dancers dance, they dance as if they aredancing in the air?"

# Performing stemming using lancaster stemmer on a sentence

print("\n2. Performing lancaster stemming on a sentence")

words = word\_tokenize(sentence, language = 'english')

for each\_word in words:

print(lancaster.stem(each\_word))

# Performing stemming using lancaster stemmer on a text file

print("\n3. Performing lancaster stemming on a text file - one sentence at a time")

# Treating the text file as a collection of sentences

reeba\_file = open("trupti.txt")

my\_lines\_list = reeba\_file.readlines()

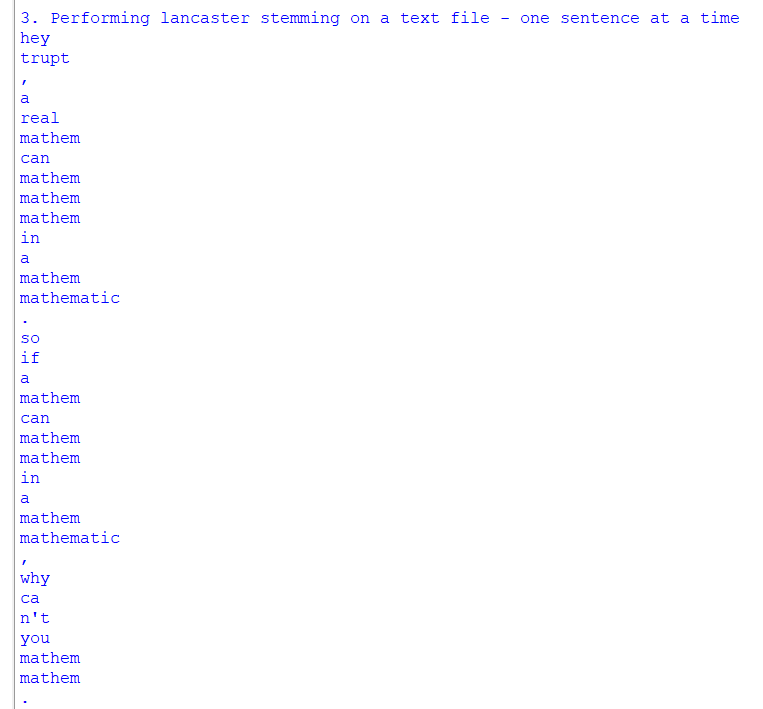
# Accessing one line at a time from the text file

words = word\_tokenize(my\_lines\_list[0], language = 'english')

for each\_word in words:

print(lancaster.stem(each\_word))

**Output :**



**c. Snowball Stemmer**

The Snowball Stemmer, also known as the Porter2 Stemmer, is an advanced and more versatile version of the original Porter Stemmer, created by Martin Porter. Introduced in 2001, it provides more consistent and maintainable stemming rules and supports multiple languages, making it a robust tool in natural language processing (NLP).

**Code :**

# Program to implement Snowball Stemmer

print("Name : Trupti Bhostekar \tRoll No : 2")

import nltk

nltk.download('punkt')

from nltk.stem.snowball import SnowballStemmer

# Defining the stemmer

snowball\_english = SnowballStemmer("english")

snowball\_english = SnowballStemmer("dutch")

# Performing stemming on one word

print("\n1. Performing snowball stemming one word")

word = snowball\_english.stem("Vibing")

print(word)

# Taking a list of english words

terms = ["trupti", "cheerful", "bravery","drawing", "satisfactorily", "publisher", "painful", "hardworking",

"keys"]

# Performing stemming using snowball stemmer on words

print("\n2. Performing snowball stemming on a set of english language words")

for each\_term in terms:

print(snowball\_english.stem(each\_term))

# Taking a list of dutch words

# trupti = trupti, bessen = berries, vriendelijkheid = friendliness, hobbelig = bumpy

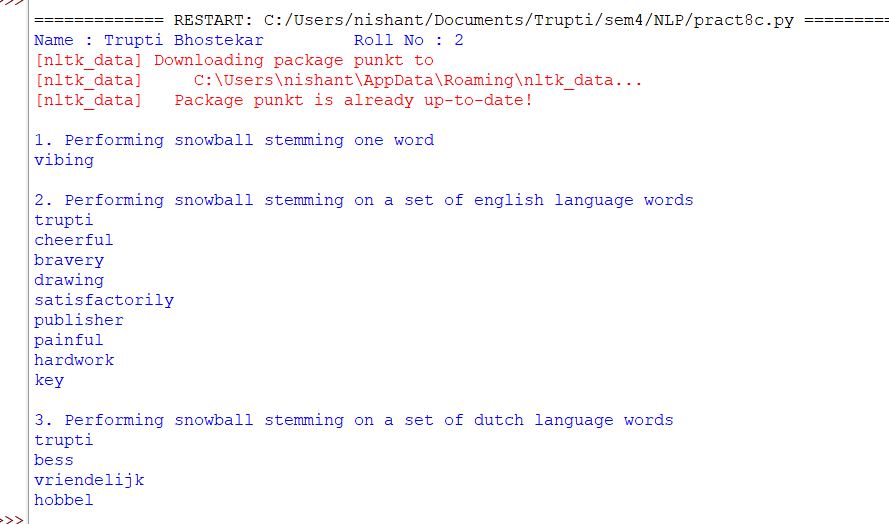
terms2 = ["trupti", "bessen", "vriendelijkheid", "hobbelig"]

print("\n3. Performing snowball stemming on a set of dutch language words")

for each\_term in terms2:

print(snowball\_english.stem(each\_term))

**Output :**



**d. RegExp Stemmer**

The RegExp Stemmer, short for Regular Expression Stemmer, is a simpler and more flexible stemming approach that uses regular expressions (regex) to define rules for reducing words to their root forms. This method relies on pattern matching and replacement techniques, making it highly customizable for specific needs. Unlike more structured algorithms like Porter or Snowball, the RegExp Stemmer is straightforward but requires careful crafting of regex patterns to ensure effectiveness.

**Code :**

# Program to implement RegExp Stemmer

print("Name : Trupti Bhostekar \tRoll No : 2")

import nltk

nltk.download('punkt')

from nltk.stem import RegexpStemmer

# Defining the stemmer

regexp = RegexpStemmer('ing$|s$|e$|able$|ment$|less$|ly$', min=4)

# Performing stemming one word

print("\n1. Performing regexp stemming on one word at a time")

print(regexp.stem('cars'))

print(regexp.stem('bee'))

print(regexp.stem('compute'))

# Taking a list of word

terms = ["truptis", "stemming", "mentally", "ease","rockstar", "frictionless", "management","flowers",

"advisable"]

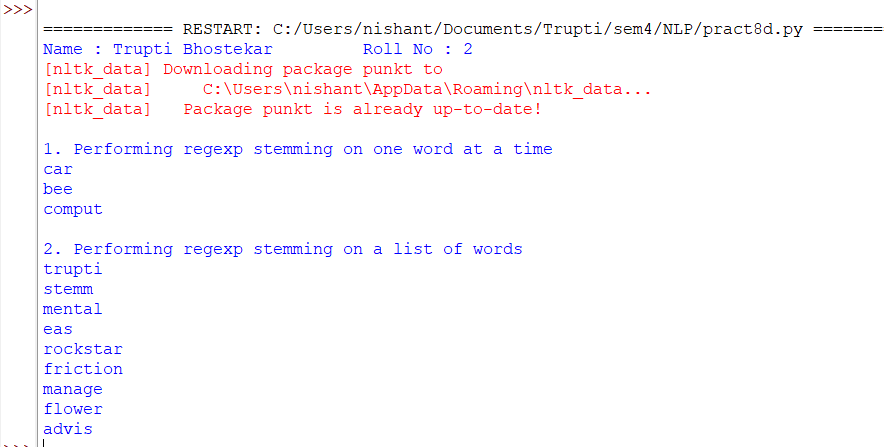
# Performing stemming using lancaster stemmer on words

print("\n2. Performing regexp stemming on a list of words")

for each\_term in terms:

print(regexp.stem(each\_term))

**Output :**



**PART B : STUDY WORDNET LEMMATIZER**

The WordNet Lemmatizer is a tool used in natural language processing (NLP) to reduce words to their base or dictionary form, known as the lemma. Unlike stemming, which often produces root forms by simply stripping suffixes, lemmatization considers the context and grammatical structure of the word to ensure the resulting lemma is a valid word.

**Code :**

# Program to implement WordNet Lemmatizer

print("Name : Trupti bhostekar \tRoll No : 2")

import nltk

nltk.download('wordnet')

nltk.download('punkt')

from nltk.tokenize import word\_tokenize

from nltk.stem import WordNetLemmatizer

# Initializing the Wordnet Lemmatizer

wordnet = WordNetLemmatizer()

# Performing WordNet lemmatization on single Words

print("\n1. Performing WordNet lemmatization on single Words")

print(wordnet.lemmatize("corpora"))

print(wordnet.lemmatize("best"))

print(wordnet.lemmatize("geese"))

print(wordnet.lemmatize("feet"))

print(wordnet.lemmatize("cacti"))

#Performing WordNet lemmatization on a sentence

print("\n2. Performing WordNet lemmatization on a sentence")

# Taking a sentence

sentence = "Heyaa trupti, how are you doing? Keep digging in for the sentences to observe lemmatization!"

# Tokenizing i.e. spliting the sentence into words

list\_words = nltk.word\_tokenize(sentence)

print("\nConverting the sentence into a list of words")

print(list\_words)

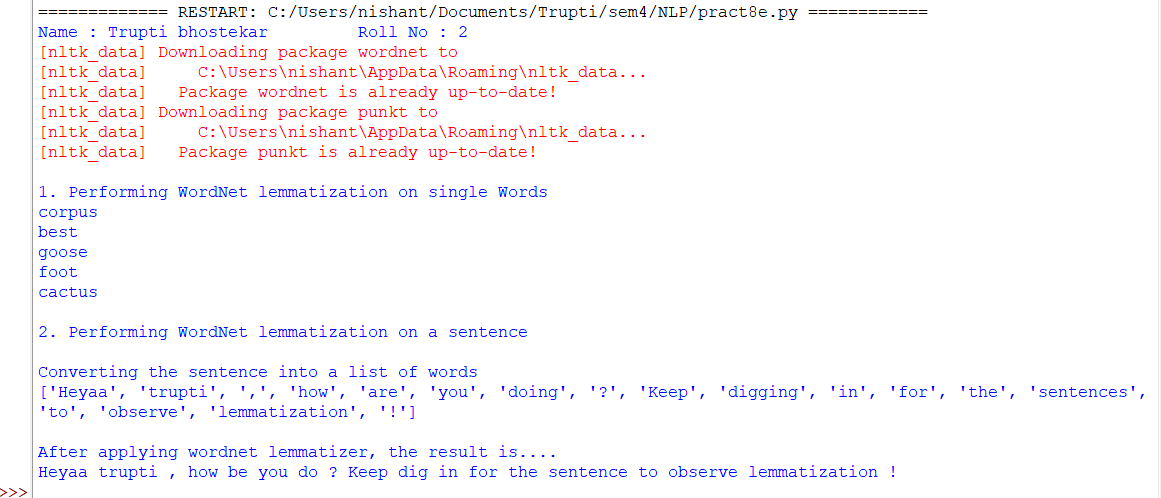
# Lemmatize list of words and join

final = ' '.join([wordnet.lemmatize(each\_word, pos = 'v') for each\_word in list\_words])

print("\nAfter applying wordnet lemmatizer, the result is....")

print(final)

**Output :**



**Practical no 8**

**Aim: Implement Naive Bayes classifier.**

Naive Bayes is a simple technique for constructing classifiers: models that assign class labels to

problem instances, represented as vectors of feature values, where the class labels are drawn from some finite set. There is not a single algorithm for training such classifiers, but a family of algorithms based on a common principle: all naive Bayes classifiers assume that the value of a particular feature is independent of the value of any other feature, given the class variable.

For example, a fruit may be considered to be an apple if it is red, round, and about 10 cm in diameter. A naive Bayes classifier considers each of these features to contribute independently to the probability that this fruit is an apple, regardless of any possible correlations between the color, roundness, and diameter features. Gaussian Naive Bayes is a variant of Naive Bayes that follows Gaussian normal distribution and supports continuous data. We have explored the idea behind Gaussian Naive Bayes along with an example. Naive Bayes are a group of supervised machine learning classification algorithms based on the Bayes theorem.

**Code :**

import sklearn

from sklearn import datasets

from sklearn.model\_selection import train\_test\_split

from sklearn.naive\_bayes import GaussianNB, MultinomialNB

from sklearn.metrics import classification\_report, confusion\_matrix, accuracy\_score

breastcancer = datasets.load\_breast\_cancer()

print("\nFeatures of breastcancer dataset : ", breastcancer.feature\_names)

print("\nLabels of breastcancer dataset : ", breastcancer.target\_names)

print("\nShape of breastcancer dataset : ", breastcancer.data.shape)

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

R = breastcancer.data

T = breastcancer.target

# Splitting the dataset into training set and testing set

Rtrain, Rtest, Ttrain, Ttest = train\_test\_split(R, T, test\_size = 0.2, random\_state = 0)

# 1. Using the Gaussian Naive Bayes Classifier

gauss = GaussianNB()

# Training the Gaussian Naive Bayes model using training set

gauss.fit(Rtrain,Ttrain)

# Making predictions using the test set

pred = gauss.predict(Rtest)

# Generating classification report of the Gaussian Naive Bayes Model

gcr = classification\_report(Ttest,pred)

print("\nClassification Report gaussian : \n", gcr)

# Generating confusion matrix of the Gaussian Naive Bayes Model

gcm = confusion\_matrix(Ttest, pred)

print("\nConfusion matrix gaussian : \n", gcm)

# Evaluating the naive bayes classifier on the basis of accuracy metric

accuracy = accuracy\_score(Ttest, pred)

print("\nAccuracy : ", accuracy \* 100)

**Output :**

