INDEX

Practical 1

Theory:

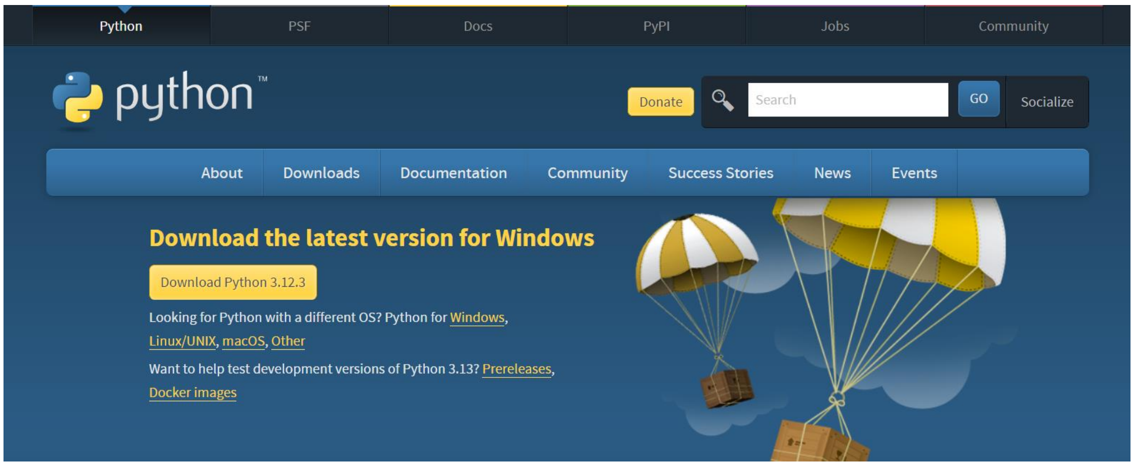
Natural Language Toolkit (NLTK) is a comprehensive Python library widely used in natural language processing (NLP) tasks. It provides a suite of libraries and programs for symbolic and statistical NLP. NLTK includes modules for tokenization, stemming, tagging, parsing, and semantic reasoning, making it a powerful tool for processing and analysing human language data. One of its key features is its extensive collection of corpora and lexical resources, which are essential for training and testing NLP models. NLTK's simplicity and ease of use make it a popular choice for both beginners and experienced researchers in the field of NLP.

NLTK's tokenization module allows for breaking text into individual words or sentences, crucial for various NLP tasks such as text analysis and information retrieval. Its stemming module enables reducing words to their base or root form, aiding in tasks like text normalization and word frequency analysis. NLTK's part-of-speech tagging functionality assigns grammatical categories to words in a text, facilitating syntactic analysis and parsing. Furthermore, NLTK supports parsing techniques like recursive descent parsing and chart parsing, enabling the extraction of structured information from text. Overall, NLTK's rich set of functionalities and resources make it a versatile and indispensable tool for natural language processing tasks.

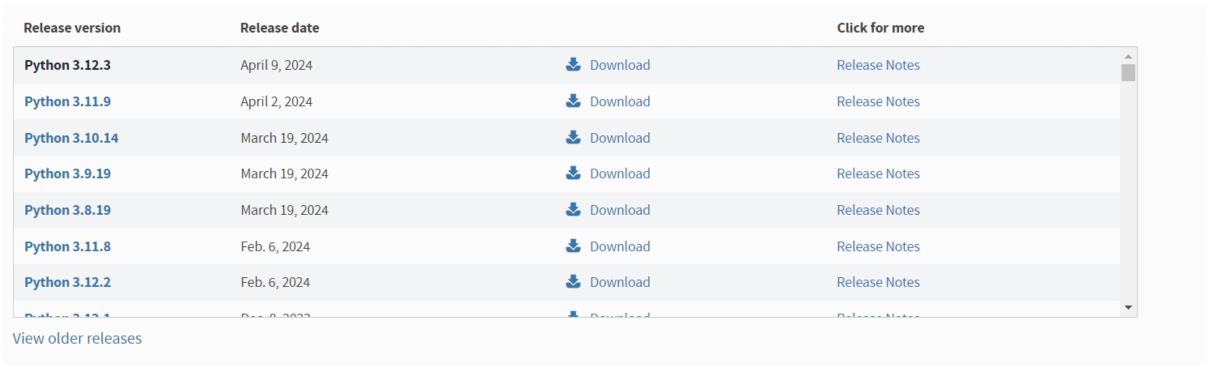
A: Install NLTK

Steps:

Step 1. Go to link https://www.python.org/downloads/, and select the latest version for windows.


Python
PSF
Docs
PyPI
Jobs
Community
python
Donate
Search
GO
Socialize
About
Downloads
Documentation
Community
Success Stories
News
Events
Download the latest version for Windows
Download Python 3.12.3
Looking for Python with a different OS? Python for Windows, Linux/UNIX, macOS, Other
Want to help test development versions of Python 3.13? Prereleases, Docker images

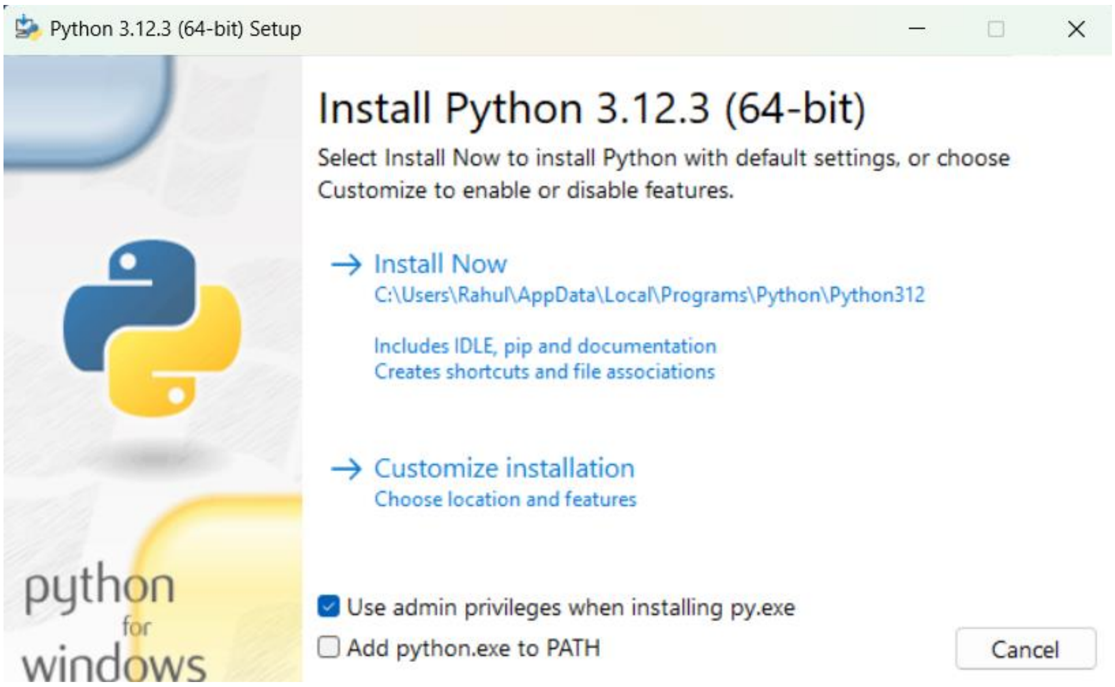
Note: If you don't want to download the latest version, you can visit the download tab and see all releases.



Release version	Release date		Click for more
Python 3.12.3	April 9, 2024	Download	Release Notes
Python 3.11.9	April 2, 2024	Download	Release Notes
Python 3.10.14	March 19, 2024	Download	Release Notes
Python 3.9.19	March 19, 2024	Download	Release Notes
Python 3.8.19	March 19, 2024	Download	Release Notes
Python 3.11.8	Feb. 6, 2024	Download	Release Notes
Python 3.12.2	Feb. 6, 2024	Download	Release Notes

Step 2. Click on Download (Python 3.12.3)

Step 3. Select Customize Installation



Python 3.12.3 (64-bit) Setup

Install Python 3.12.3 (64-bit)

Select Install Now to install Python with default settings, or choose Customize to enable or disable features.

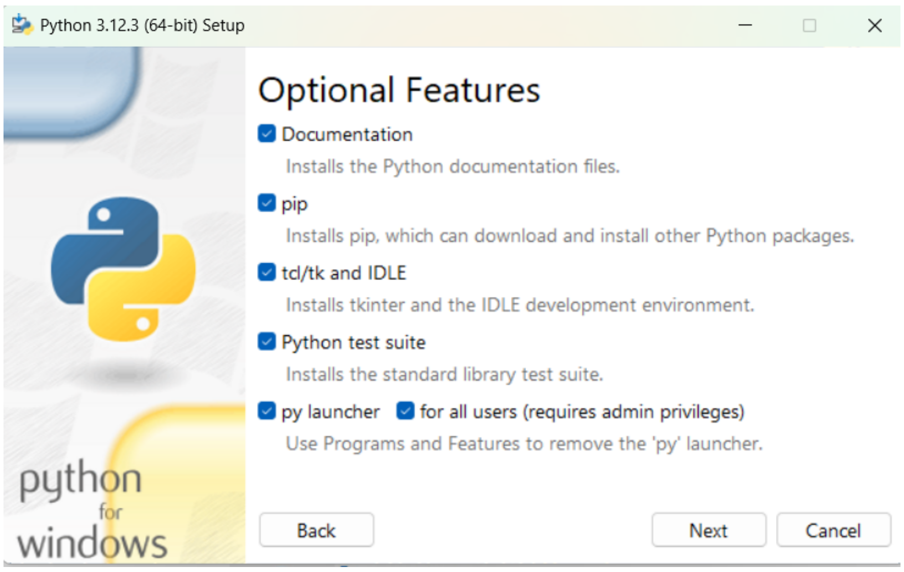
→ Install Now
C:\Users\Rahul\AppData\Local\Programs\Python\Python312

Includes IDLE, pip and documentation
Creates shortcuts and file associations

→ Customize installation
Choose location and features

☑ Use admin privileges when installing py.exe
☐ Add python.exe to PATH

Step 4. Click Next



Python 3.12.3 (64-bit) Setup

Optional Features

☑ Documentation
Installs the Python documentation files.

☑ pip
Installs pip, which can download and install other Python packages.

☑ tcl/tk and IDLE
Installs tkinter and the IDLE development environment.

☑ Python test suite
Installs the standard library test suite.

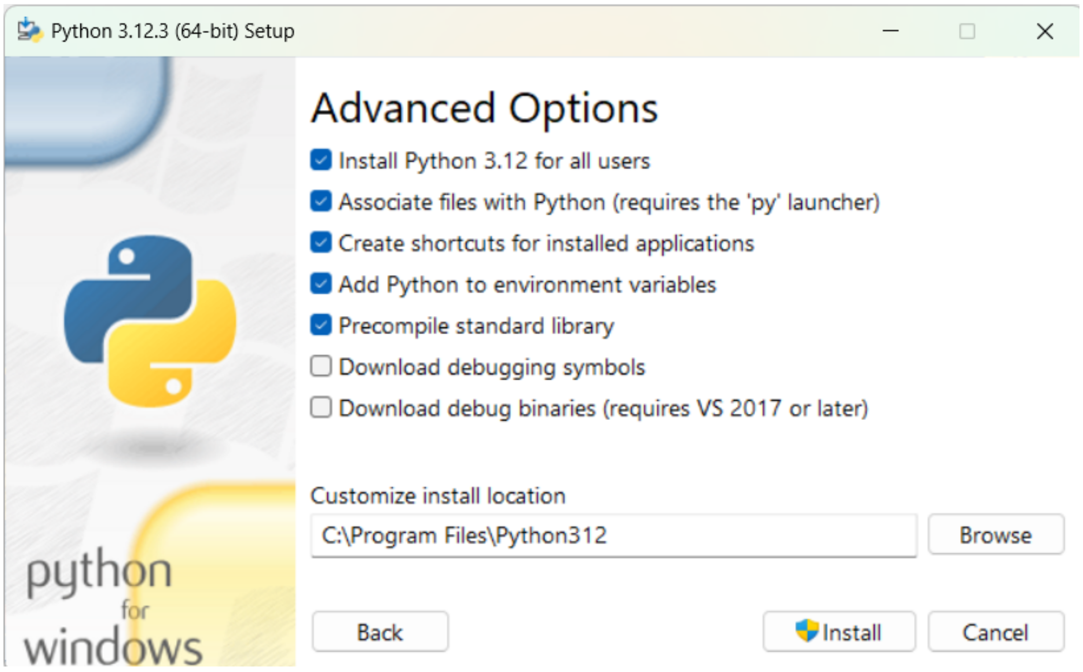
☑ py launcher ☑ for all users (requires admin privileges)
Use Programs and Features to remove the 'py' launcher.

Step 5. In next screen

1. Select the advanced options

2. Give a Custom install location.

3. Click Install


Python 3.12.3 (64-bit) Setup

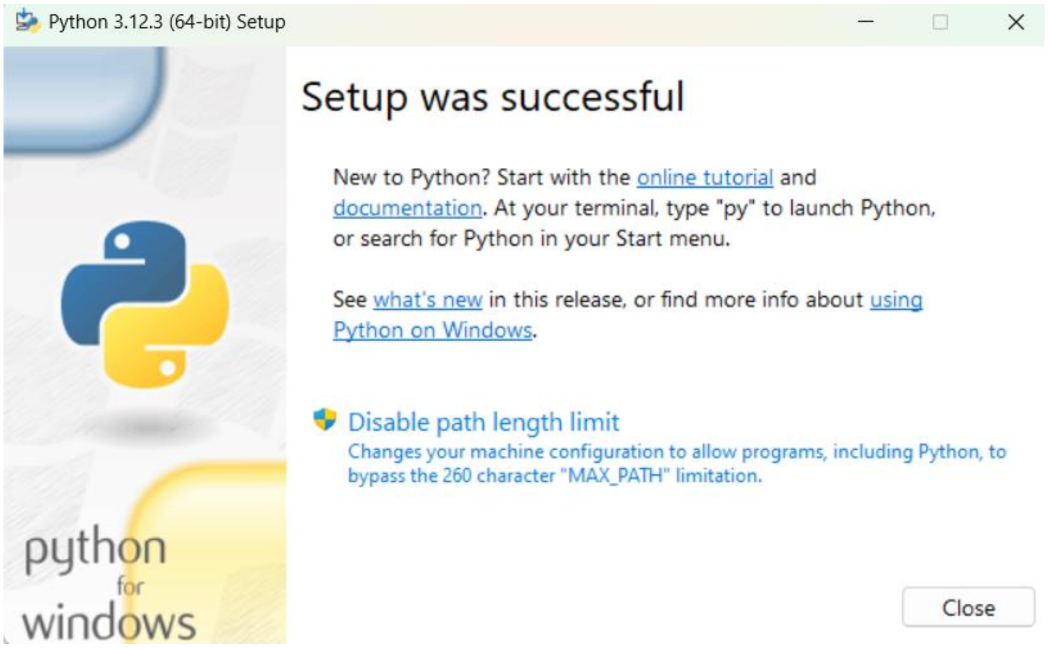
Advanced Options

- [x] Install Python 3.12 for all users
- [x] Associate files with Python (requires the 'py' launcher)
- [x] Create shortcuts for installed applications
- [x] Add Python to environment variables
- [x] Precompile standard library
- [ ] Download debugging symbols
- [ ] Download debug binaries (requires VS 2017 or later)

Customize install location

C:\Program Files\Python312

Step 6. Click Close button once install is done.


Python 3.12.3 (64-bit) Setup

Setup was successful

New to Python? Start with the online tutorial and documentation. At your terminal, type "py" to launch Python, or search for Python in your Start menu.

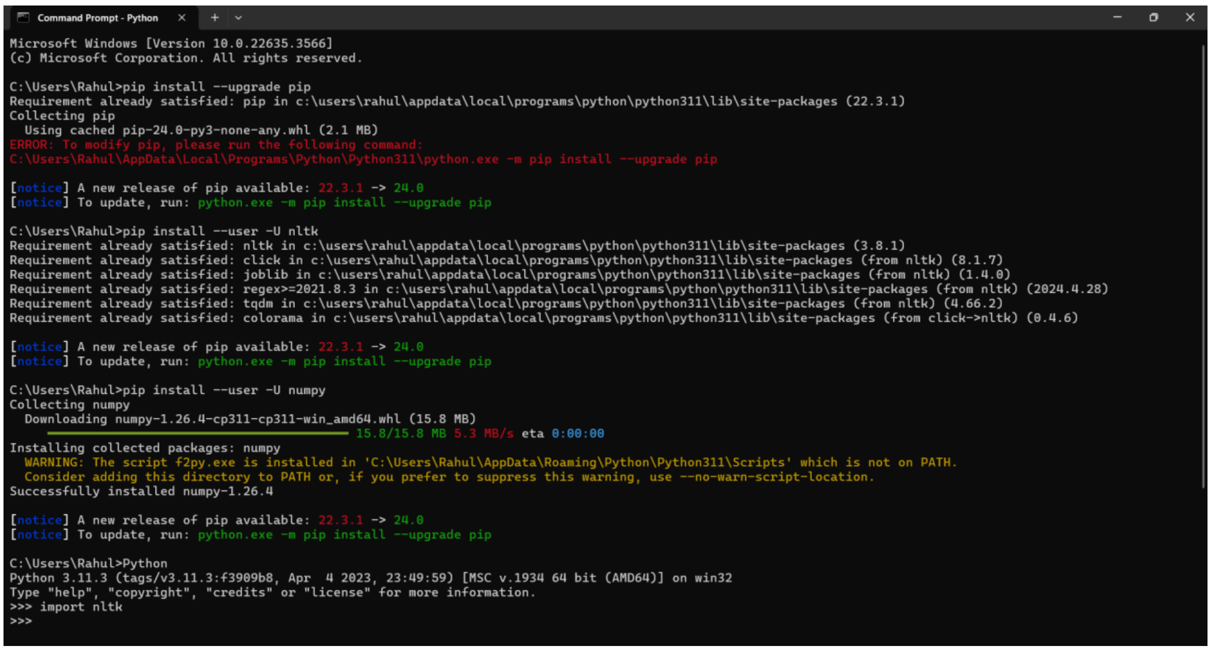
See what's new in this release, or find more info about using Python on Windows.

Disable path length limit
Changes your machine configuration to allow programs, including Python, to bypass the 260 character "MAX_PATH" limitation.

Step 7) open command prompt window and run the following commands:

```
pip install --upgrade pip
pip install --user -U nltk
pip install --user -U numpy
Type python in command line and press Enter
>>> import nltk
```

Output:



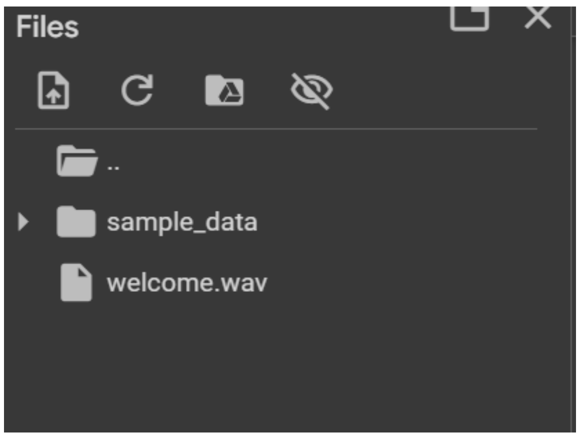
```
Microsoft Windows [Version 10.0.22635.3566]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Rahul>pip install --upgrade pip
Requirement already satisfied: pip in c:\users\rahul\appdata\local\programs\python\python311\lib\site-packages (22.3.1)
Collecting pip
  Using cached pip-24.0-py3-none-any.whl (2.1 MB)
ERROR: To modify pip, please run the following command:
C:\Users\Rahul\AppData\Local\Programs\Python\Python311\python.exe -m pip install --upgrade pip

[notice] A new release of pip available: 22.3.1 -> 24.0
[notice] To update, run: python.exe -m pip install --upgrade pip

C:\Users\Rahul>pip install --user -U nltk
Requirement already satisfied: nltk in c:\users\rahul\appdata\local\programs\python\python311\lib\site-packages (3.8.1)
Requirement already satisfied: click in c:\users\rahul\appdata\local\programs\python\python311\lib\site-packages (from nltk) (8.1.7)
Requirement already satisfied: joblib in c:\users\rahul\appdata\local\programs\python\python311\lib\site-packages (from nltk) (1.4.0)
Requirement already satisfied: regex>=2021.8.3 in c:\users\rahul\appdata\local\programs\python\python311\lib\site-packages (from nltk) (2024.4.28)
Requirement already satisfied: tqdm in c:\users\rahul\appdata\local\programs\python\python311\lib\site-packages (from nltk) (4.66.2)
Requirement already satisfied: colorama in c:\users\rahul\appdata\local\programs\python\python311\lib\site-packages (from click->nltk) (0.4.6)

[notice] A new release of pip available: 22.3.1 -> 24.0
[notice] To update, run: python.exe -m pip install --upgrade pip

C:\Users\Rahul>pip install --user -U numpy
Collecting numpy
  Downloading numpy-1.26.4-cp311-cp311-win_amd64.whl (15.8 MB)
     ---------------------------------------- 15.8/15.8 MB 5.3 MB/s eta 0:00:00
Installing collected packages: numpy
  WARNING: The script f2py.exe is installed in 'C:\Users\Rahul\AppData\Roaming\Python\Python311\Scripts' which is not on PATH.
  Consider adding this directory to PATH or, if you prefer to suppress this warning, use --no-warn-script-location.
Successfully installed numpy-1.26.4

[notice] A new release of pip available: 22.3.1 -> 24.0
[notice] To update, run: python.exe -m pip install --upgrade pip

C:\Users\Rahul>Python
Python 3.11.3 (tags/v3.11.3:f3909b8, Apr  4 2023, 23:49:59) [MSC v.1934 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import nltk
>>>
```

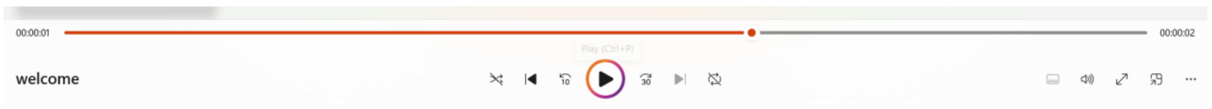
B. Convert the given text to speech.

Code: -

```
!pip install gtts
from gtts import gTTS
# This module is imported so that we can
# play the converted audio
import os
# The text that you want to convert to audio
mytext = 'Welcome to Natural Language Processing'
# Language in which you want to convert
language = 'en'
# Passing the text and language to the engine,
# here we have marked slow=False. Which tells
# the module that the converted audio should
# have a high speed
myobj = gTTS(text=mytext, lang=language, slow=False)
# Saving the converted audio in a mp3 file named
# welcome
myobj.save("welcome.wav")
# Playing the converted file
os.system("welcome.wav")
```

Output: -



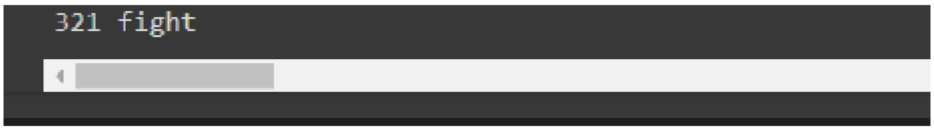


C. Convert audio file Speech to Text.

Code: -

```
!pip3 install SpeechRecognition pydub
!file audio.mp3
!ffmpeg -i audio.mp3 audio.wav
import speech_recognition as sr
# initialize the recognizer
r = sr.Recognizer()
# open the file
with sr.AudioFile('/content/audio.wav') 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: -



```
321 fight
```

Practical 2

Theory:

The study of various corpora such as Brown, Inaugural, Reuters, and UDHR (Universal Declaration of Human Rights) offers invaluable insights into language diversity, usage patterns, and thematic content. NLTK provides several methods for exploring these corpora, each offering unique perspectives on the data.

Using the "fields" method, one can access specific metadata fields associated with each document in the corpora, such as genre, author, or publication date. This allows researchers to analyze how language usage varies across different contexts and time periods.

The "raw" method provides access to the raw text of the documents in the corpora, enabling basic text processing tasks such as tokenization and stemming. Researchers can use this method to perform statistical analyses on word frequency, distribution, and other linguistic properties.

The "words" method breaks down the text into individual words, allowing for more granular analysis of vocabulary usage and lexical diversity. This method is particularly useful for studying language variation and evolution over time.

The "sents" method splits the text into sentences, facilitating syntactic analysis and parsing. Researchers can use this method to study sentence structure, grammar, and syntactic patterns across different genres and domains.

Finally, the "categories" method categorizes documents in the corpora based on predefined labels or tags, such as topic or genre. This method enables researchers to study thematic trends, discourse patterns, and domain-specific language usage within the corpora.

By leveraging these methods in NLTK, researchers can gain deeper insights into the linguistic characteristics and usage patterns of various corpora, thereby advancing our understanding of natural language processing and its applications.

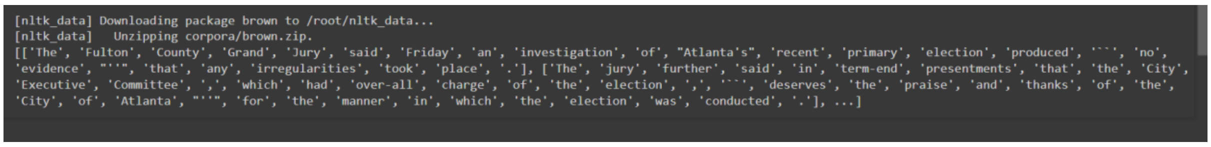
A. Study of various Corpus – Brown, Inaugural, Reuters, udhr with various methods like fields, raw, words, sents, categories.

Code:

```
# Brown corpus
from nltk.corpus import brown
import nltk
nltk.download('brown')
brown.categories()
brown.words(categories='news')
brown.words(fileids=['cg22'])
brown.sents(categories=['news', 'editorial', 'reviews'])

# reuters corpus
from nltk.corpus import reuters
import nltk
nltk.download('reuters')
reuters.fileids()
reuters.categories()
reuters.categories('training/9865')
reuters.fileids('barley')
reuters.words('training/9865')[:14]

# inaugral corpus
from nltk.corpus import inaugural
import nltk
nltk.download('inaugural')
inaugural.fileids()
[fileid[:4] for fileid in inaugural.fileids()]
```

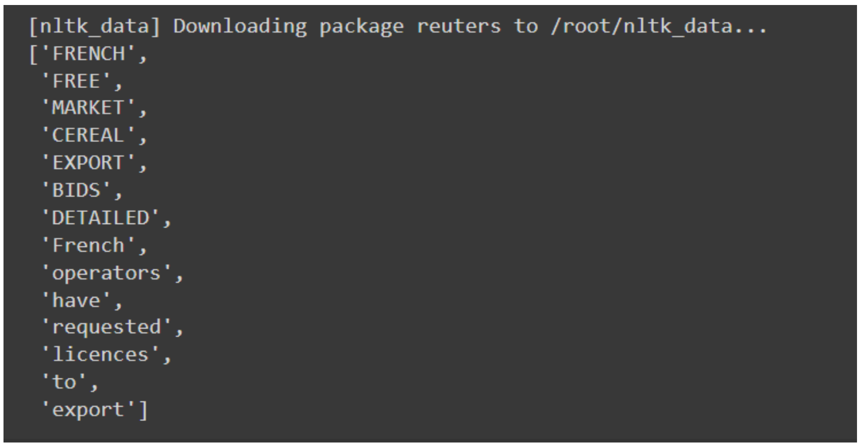
Output:

Brown corpus:



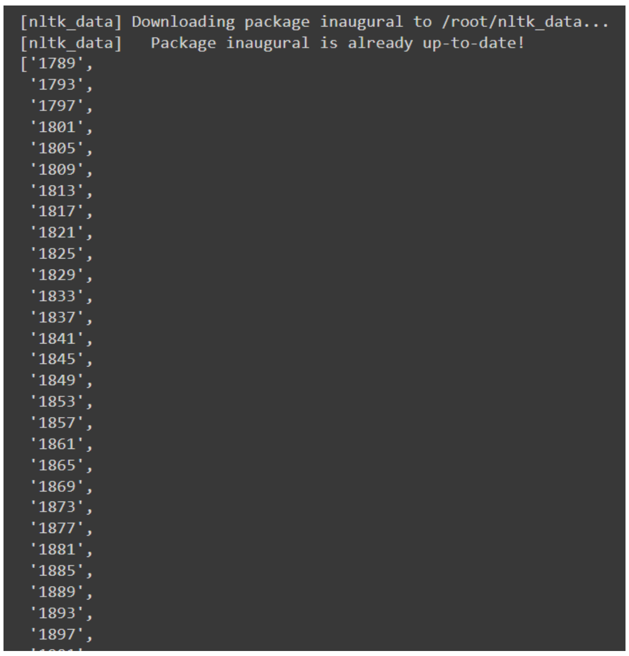
```
[nltk_data] Downloading package brown to /root/nltk_data...
[nltk_data]   Unzipping corpora/brown.zip.
[['The', 'Fulton', 'County', 'Grand', 'Jury', 'said', 'Friday', 'an', 'investigation', 'of', "Atlanta's", 'recent', 'primary', 'election', 'produced', '``', 'no',
'evidence', "''", 'that', 'any', 'irregularities', 'took', 'place', '.'], ['The', 'jury', 'further', 'said', 'in', 'term-end', 'presentments', 'that', 'the', 'City',
'Executive', 'Committee', ',', 'which', 'had', 'over-all', 'charge', 'of', 'the', 'election', ',', '``', 'deserves', 'the', 'praise', 'and', 'thanks', 'of', 'the',
'City', 'of', 'Atlanta', "''", 'for', 'the', 'manner', 'in', 'which', 'the', 'election', 'was', 'conducted', '.'], ...]
```

Reuters corpus:

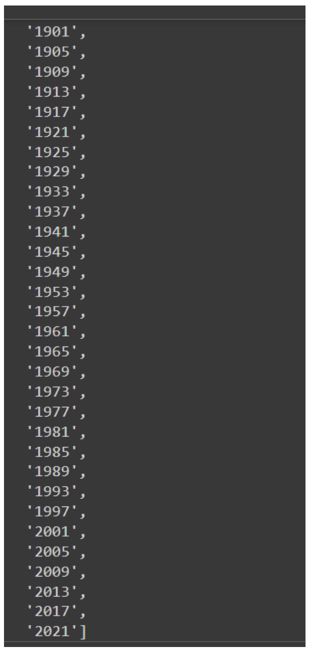


```
[nltk_data] Downloading package reuters to /root/nltk_data...
['FRENCH',
 'FREE',
 'MARKET',
 'CEREAL',
 'EXPORT',
 'BIDS',
 'DETAILED',
 'French',
 'operators',
 'have',
 'requested',
 'licences',
 'to',
 'export']
```

Inaugral corpus:



```
[nltk_data] Downloading package inaugural to /root/nltk_data...
[nltk_data]   Package inaugural is already up-to-date!
['1789',
 '1793',
 '1797',
 '1801',
 '1805',
 '1809',
 '1813',
 '1817',
 '1821',
 '1825',
 '1829',
 '1833',
 '1837',
 '1841',
 '1845',
 '1849',
 '1853',
 '1857',
 '1861',
 '1865',
 '1869',
 '1873',
 '1877',
 '1881',
 '1885',
 '1889',
 '1893',
 '1897',
```



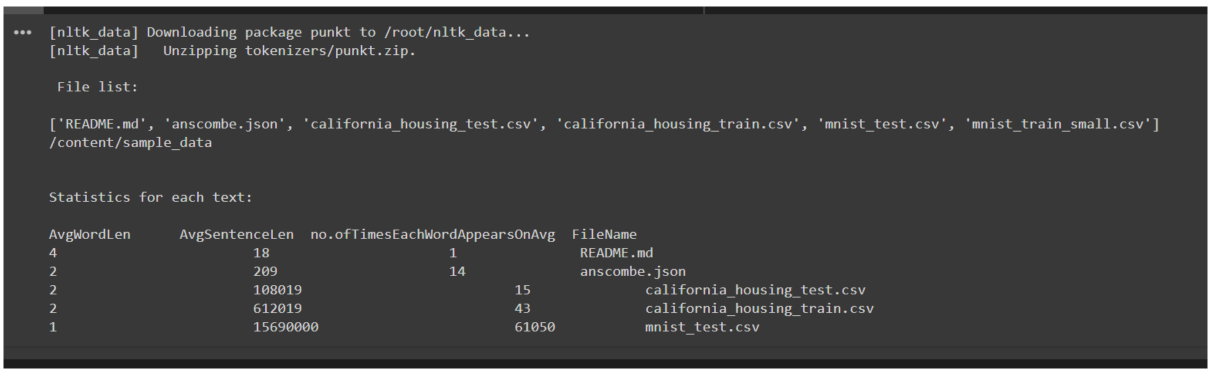
```
 '1901',
 '1905',
 '1909',
 '1913',
 '1917',
 '1921',
 '1925',
 '1929',
 '1933',
 '1937',
 '1941',
 '1945',
 '1949',
 '1953',
 '1957',
 '1961',
 '1965',
 '1969',
 '1973',
 '1977',
 '1981',
 '1985',
 '1989',
 '1993',
 '1997',
 '2001',
 '2005',
 '2009',
 '2013',
 '2017',
 '2021']
```

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

Code:

```
import nltk
nltk.download('punkt')
from nltk.corpus import PlaintextCorpusReader
corpus_root = '/content/sample_data'
filelist = PlaintextCorpusReader(corpus_root, '.*')
print ('\n File list: \n')
print (filelist.fileids())
print (filelist.root)
print ('\n\nStatistics for each text:\n')
print
('AvgWordLen\tAvgSentenceLen\tno.ofTimesEachWordAppearsOnAvg\tFileName')
for fileid in filelist.fileids():
 num_chars = len(filelist.raw(fileid))
 num_words = len(filelist.words(fileid))
 num_sents = len(filelist.sents(fileid))
 num_vocab = len(set([w.lower() for w in filelist.words(fileid)]))
 print(int(num_chars/num_words),'\t\t\t',
int(num_words/num_sents),'\t\t\t',
int(num_words/num_vocab),'\t\t', fileid)
```

Output:



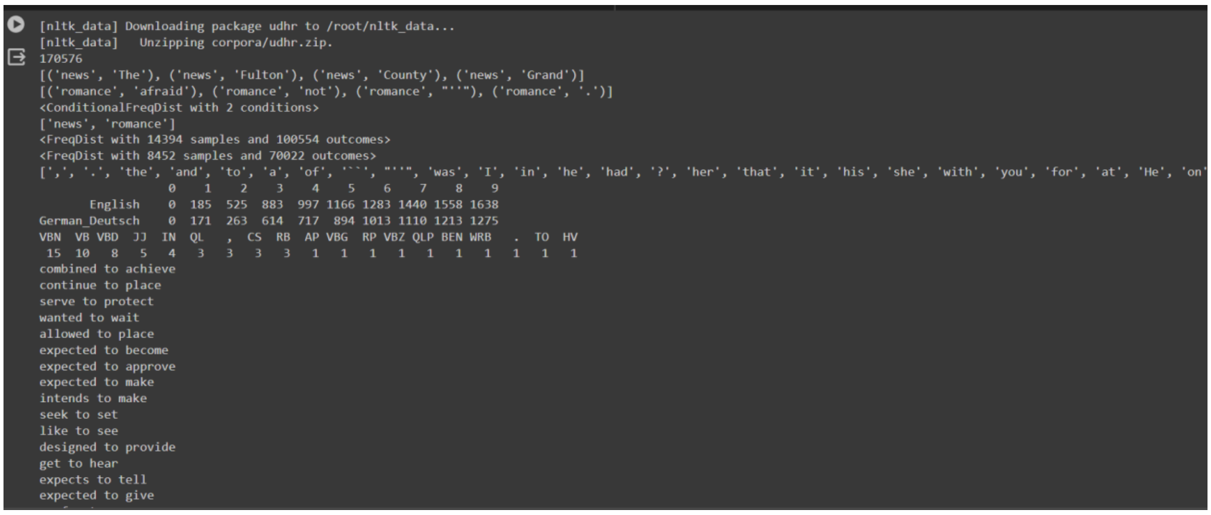
```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt.zip.

 File list:

['README.md', 'anscombe.json', 'california_housing_test.csv', 'california_housing_train.csv', 'mnist_test.csv', 'mnist_train_small.csv']
/content/sample_data


Statistics for each text:

AvgWordLen      AvgSentenceLen  no.ofTimesEachWordAppearsOnAvg  FileName
4                        18                     1                README.md
2                        209                    14               anscombe.json
2                        108019                         15               california_housing_test.csv
2                        612019                         43               california_housing_train.csv
1                        15690000                       61050            mnist_test.csv
```

C. Study Conditional frequency distributions. Study of tagged corpora with methods like tagged_sents, and tagged_words.

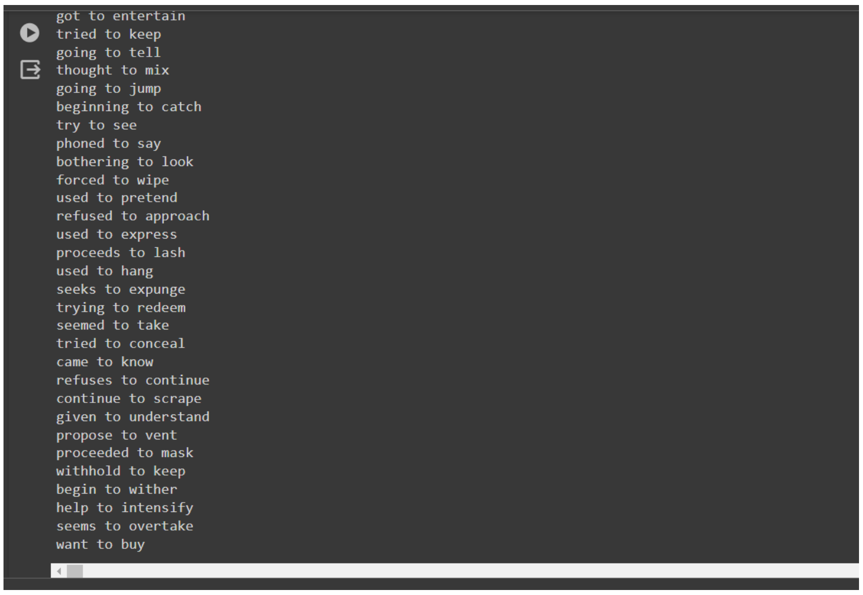
Code:

```
text = ['The', 'Fulton', 'County', 'Grand', 'Jury', 'said', ...]
pairs = [('news', 'The'), ('news', 'Fulton'), ('news', 'County'), ...]
import nltk
nltk.download('udhr')
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)
# study of tagged corpora
import nltk
brown_lrnd_tagged = brown.tagged_words(categories='learned')
tags = [b[1] for (a, b) in nltk.bigrams(brown_lrnd_tagged) if a[0] ==
'often']
fd = nltk.FreqDist(tags)
fd.tabulate()
from nltk.corpus import brown
def process(sentence):
 for (w1,t1), (w2,t2), (w3,t3) in nltk.trigrams(sentence):
  if (t1.startswith('V') and t2 == 'TO' and t3.startswith('V')):
    print(w1, w2, w3)
for tagged_sent in brown.tagged_sents():
  process(tagged_sent)
```

Output:



```
[nltk_data] Downloading package udhr to /root/nltk_data...
[nltk_data]   Unzipping corpora/udhr.zip.
170576
[('news', 'The'), ('news', 'Fulton'), ('news', 'County'), ('news', 'Grand')]
[('romance', 'afraid'), ('romance', 'not'), ('romance', "''"), ('romance', '.')]
<ConditionalFreqDist with 2 conditions>
['news', 'romance']
<FreqDist with 14394 samples and 100554 outcomes>
<FreqDist with 8452 samples and 70022 outcomes>
[',', '.', 'the', 'and', 'to', 'a', 'of', '``', "''", 'was', 'I', 'in', 'he', 'had', '?', 'her', 'that', 'it', 'his', 'she', 'with', 'you', 'for', 'at', 'He', 'on'
                 0    1    2    3    4    5    6    7    8    9
       English    0  185  525  883  997 1166 1283 1440 1558 1638
German_Deutsch    0  171  263  614  717  894 1013 1110 1213 1275
VBN  VB VBD  JJ  IN  QL   ,  CS  RB  AP VBG  RP VBZ QLP BEN WRB   .  TO  HV
 15  10   8   5   4   3   3   3   3   1   1   1   1   1   1   1   1   1   1
combined to achieve
continue to place
serve to protect
wanted to wait
allowed to place
expected to become
expected to approve
expected to make
intends to make
seek to set
like to see
designed to provide
get to hear
expects to tell
expected to give
```



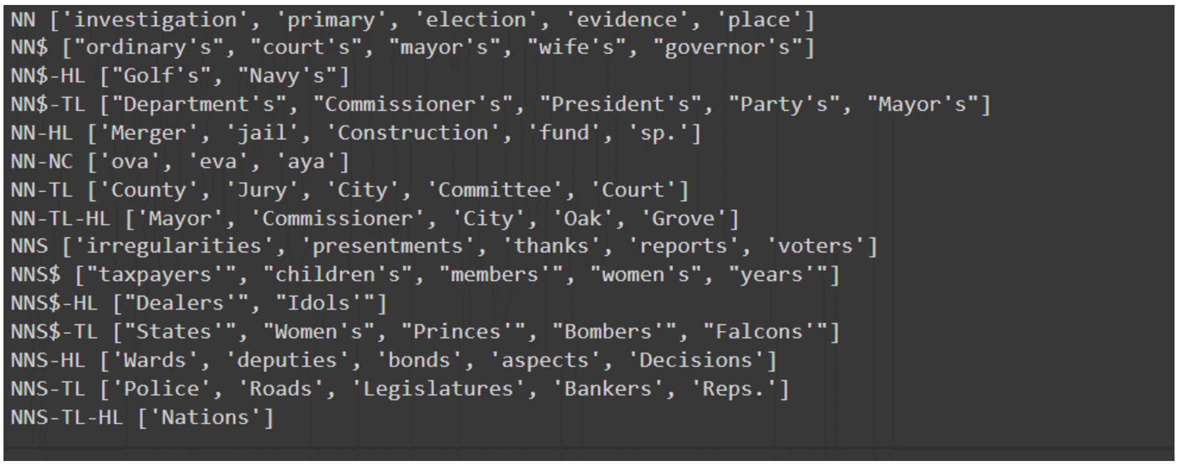
```
got to entertain
tried to keep
going to tell
thought to mix
going to jump
beginning to catch
try to see
phoned to say
bothering to look
forced to wipe
used to pretend
refused to approach
used to express
proceeds to lash
used to hang
seeks to expunge
trying to redeem
seemed to take
tried to conceal
came to know
refuses to continue
continue to scrape
given to understand
propose to vent
proceeded to mask
withhold to keep
begin to wither
help to intensify
seems to overtake
want to buy
```

D. Write a program to find the most frequent noun tags.

Code:

```
import nltk
nltk.download('all')
def findtags(tag_prefix, tagged_text):
 cfd = nltk.ConditionalFreqDist((tag, word) for (word, tag) in tagged_text
 if tag.startswith(tag_prefix))
 return dict((tag, list(cfd[tag].keys())[:5]) for tag in cfd.conditions())
tagdict = findtags('NN', nltk.corpus.brown.tagged_words(categories='news'))
for tag in sorted(tagdict):
  print(tag, tagdict[tag])
```

Output:



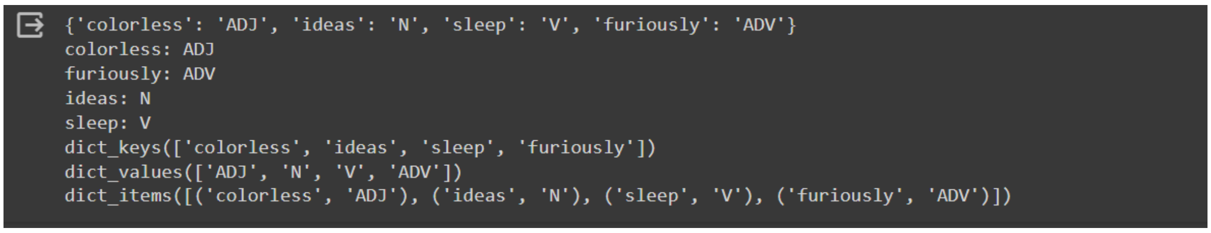
```
NN ['investigation', 'primary', 'election', 'evidence', 'place']
NN$ ["ordinary's", "court's", "mayor's", "wife's", "governor's"]
NN$-HL ["Golf's", "Navy's"]
NN$-TL ["Department's", "Commissioner's", "President's", "Party's", "Mayor's"]
NN-HL ['Merger', 'jail', 'Construction', 'fund', 'sp.']
NN-NC ['ova', 'eva', 'aya']
NN-TL ['County', 'Jury', 'City', 'Committee', 'Court']
NN-TL-HL ['Mayor', 'Commissioner', 'City', 'Oak', 'Grove']
NNS ['irregularities', 'presentments', 'thanks', 'reports', 'voters']
NNS$ ["taxpayers'", "children's", "members'", "women's", "years'"]
NNS$-HL ["Dealers'", "Idols'"]
NNS$-TL ["States'", "Women's", "Princes'", "Bombers'", "Falcons'"]
NNS-HL ['Wards', 'deputies', 'bonds', 'aspects', 'Decisions']
NNS-TL ['Police', 'Roads', 'Legislatures', 'Bankers', 'Reps.']
NNS-TL-HL ['Nations']
```

E. Map Words to Properties Using Python Dictionaries.

Code:

```
pos = {}
pos['colorless'] = 'ADJ'
pos['ideas'] = 'N'
pos['sleep'] = 'V'
pos['furiously'] = 'ADV'
print(pos)
pos['ideas']
pos['colorless']
list(pos)
sorted(pos)
[w for w in pos if w.endswith('s')]
for word in sorted(pos):
  print(word + ":", pos[word])
print(pos.keys())
print(pos.values())
print(pos.items())
```

Output:



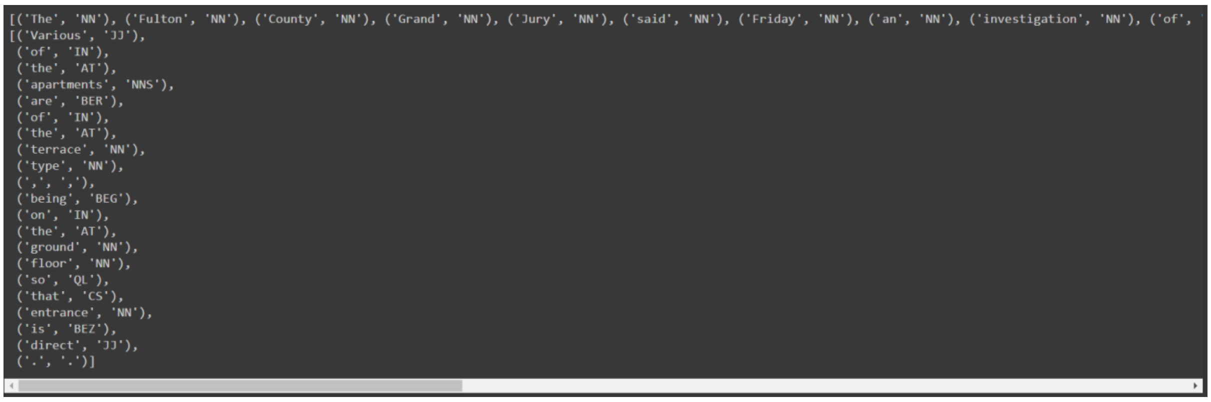
```
{'colorless': 'ADJ', 'ideas': 'N', 'sleep': 'V', 'furiously': 'ADV'}
colorless: ADJ
furiously: ADV
ideas: N
sleep: V
dict_keys(['colorless', 'ideas', 'sleep', 'furiously'])
dict_values(['ADJ', 'N', 'V', 'ADV'])
dict_items([('colorless', 'ADJ'), ('ideas', 'N'), ('sleep', 'V'), ('furiously', 'ADV')])
```

F. Study DefaultTagger, Regular expression tagger, UnigramTagger.

Code:

```
# default tagger
import nltk
from nltk.tag import DefaultTagger
raw = 'I do like to go to gym everyday!'
tokens = nltk.word_tokenize(raw)
default_tagger = nltk.DefaultTagger('NN')
default_tagger.tag(tokens)

#regular expression tagger
from nltk.corpus import brown
from nltk.tag import RegexpTagger
test_sent = brown.sents(categories='news')[0]
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)
print(regexp_tagger.tag(test_sent))

# Unigram tagger
from nltk.corpus import brown
brown_tagged_sents = brown.tagged_sents(categories='news')
brown_sents = brown.sents(categories='news')
unigram_tagger = nltk.UnigramTagger(brown_tagged_sents)
unigram_tagger.tag(brown_sents[2007])
```

Output:



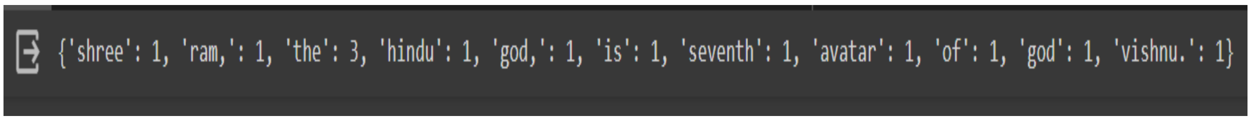
```
[('The', 'NN'), ('Fulton', 'NN'), ('County', 'NN'), ('Grand', 'NN'), ('Jury', 'NN'), ('said', 'NN'), ('Friday', 'NN'), ('an', 'NN'), ('investigation', 'NN'), ('of',
[('Various', 'JJ'),
 ('of', 'IN'),
 ('the', 'AT'),
 ('apartments', 'NNS'),
 ('are', 'BER'),
 ('of', 'IN'),
 ('the', 'AT'),
 ('terrace', 'NN'),
 ('type', 'NN'),
 (',', ','),
 ('being', 'BEG'),
 ('on', 'IN'),
 ('the', 'AT'),
 ('ground', 'NN'),
 ('floor', 'NN'),
 ('so', 'QL'),
 ('that', 'CS'),
 ('entrance', 'NN'),
 ('is', 'BEZ'),
 ('direct', 'JJ'),
 ('.', '.')]
```

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:

```
def word_count(str):
    counts = dict()
    str = str.lower()
    words = str.split()
    for word in words:
        if word in counts:
            counts[word] += 1
        else:
            counts[word] = 1
    return counts
print(word_count('Shree Ram, the Hindu god, is the seventh avatar of the god
Vishnu.'))
```

Output:



```
{'shree': 1, 'ram,': 1, 'the': 3, 'hindu': 1, 'god,': 1, 'is': 1, 'seventh': 1, 'avatar': 1, 'of': 1, 'god': 1, 'vishnu.': 1}
```

Practical 3

Theory:

Studying WordNet, a lexical database of the English language, involves exploring its rich collection of synsets (sets of synonyms), definitions, examples, and antonyms for words. WordNet provides a structured and comprehensive resource for understanding word meanings and relationships.

Synsets are sets of synonymous words (or lemmas) that represent a single concept or meaning. Each synset in WordNet is associated with a definition that describes the meaning of the concept.

WordNet provides example sentences to illustrate the usage of words in different contexts. WordNet also includes antonym relations between words.

A lemma in WordNet refers to the base form or dictionary form of a word. It represents all the different inflected forms of a word. Each synset in WordNet consists of one or more lemmas representing different forms of the same concept.

Hyponyms are words or concepts that are more specific than a given word or concept. In WordNet, hyponyms form a hierarchical structure where a hypernym (more general term) is connected to one or more hyponyms (more specific terms).

Hypernyms are words or concepts that are more general than a given word or concept. In WordNet, hypernyms also form a hierarchical structure where a hyponym (more specific term) is connected to one or more hypernyms (more general terms).

Entailments represent logical relationships between verbs where the truth of one action implies the truth of another action. In WordNet, entailments are represented as relations between verb synsets.

WordNet might not contain antonyms for every sense of a word, so the list of antonyms might be empty for some synsets. Additionally, a word can have multiple senses (synsets) with different meanings, so you might get different sets of synonyms and antonyms for each sense.

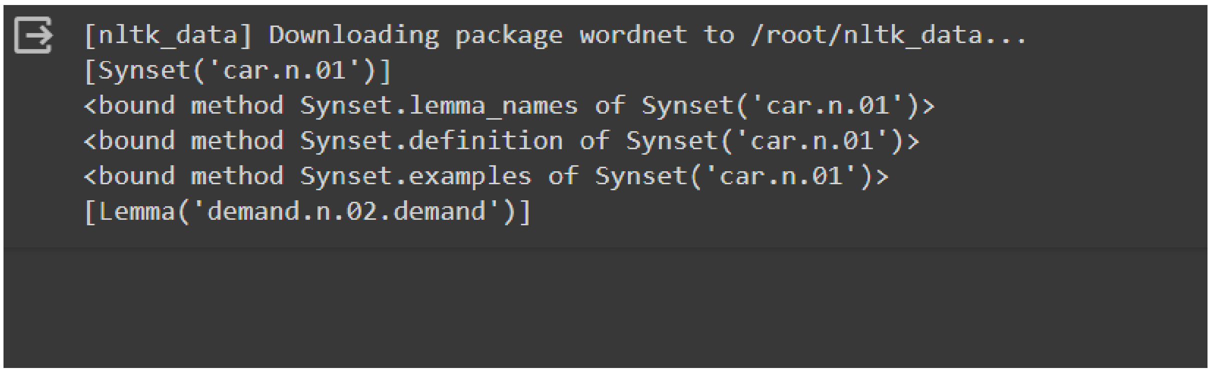
To compare two nouns using NLTK, we can use WordNet to find the shortest path between the synsets (sets of synonyms) of the two nouns in the WordNet hypernym/hyponym hierarchy. The shortest path distance gives us an indication of how closely related the two nouns are in terms of their semantic hierarchy. This approach gives us a quantitative measure of how closely related the two nouns are in terms of their semantic hierarchy in WordNet.

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

Code: -

```
import nltk
nltk.download('wordnet')
from nltk.corpus import wordnet as wn
print(wn.synsets('motorcar'))
print(wn.synset('car.n.01').lemma_names)
print(wn.synset('car.n.01').definition)
print(wn.synset('car.n.01').examples)
print(wn.lemma('supply.n.02.supply').antonyms())
```

Output: -



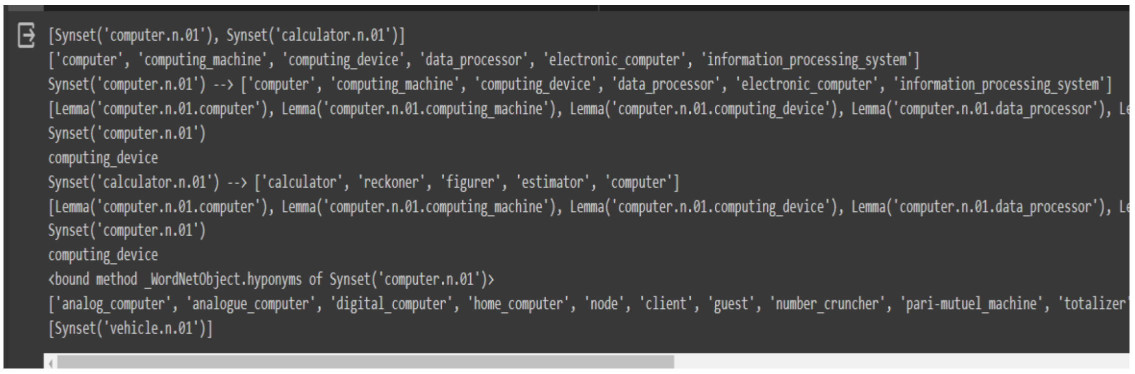
```
[nltk_data] Downloading package wordnet to /root/nltk_data...
[Synset('car.n.01')]
<bound method Synset.lemma_names of Synset('car.n.01')>
<bound method Synset.definition of Synset('car.n.01')>
<bound method Synset.examples of Synset('car.n.01')>
[Lemma('demand.n.02.demand')]
```

B. Study lemmas, hyponyms, hypernyms, entailments.

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: -



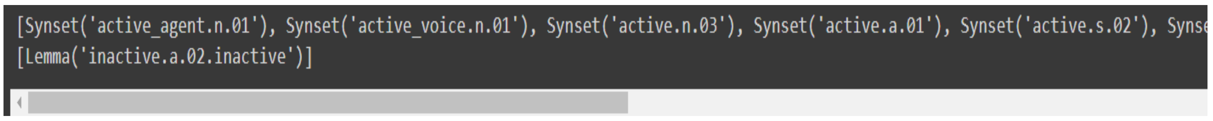
```
[Synset('computer.n.01'), Synset('calculator.n.01')]
['computer', 'computing_machine', 'computing_device', 'data_processor', 'electronic_computer', 'information_processing_system']
Synset('computer.n.01') --> ['computer', 'computing_machine', 'computing_device', 'data_processor', 'electronic_computer', 'information_processing_system']
[Lemma('computer.n.01.computer'), Lemma('computer.n.01.computing_machine'), Lemma('computer.n.01.computing_device'), Lemma('computer.n.01.data_processor'), Le
Synset('computer.n.01')
computing_device
Synset('calculator.n.01') --> ['calculator', 'reckoner', 'figurer', 'estimator', 'computer']
[Lemma('computer.n.01.computer'), Lemma('computer.n.01.computing_machine'), Lemma('computer.n.01.computing_device'), Lemma('computer.n.01.data_processor'), Le
Synset('computer.n.01')
computing_device
<bound method _WordNetObject.hyponyms of Synset('computer.n.01')>
['analog_computer', 'analogue_computer', 'digital_computer', 'home_computer', 'node', 'client', 'guest', 'number_cruncher', 'pari-mutuel_machine', 'totalizer'
[Synset('vehicle.n.01')]
```

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

Code: -

```
from nltk.corpus import wordnet
print( wordnet.synsets("active"))
print(wordnet.lemma('active.a.01.active').antonyms())
```

Output: -



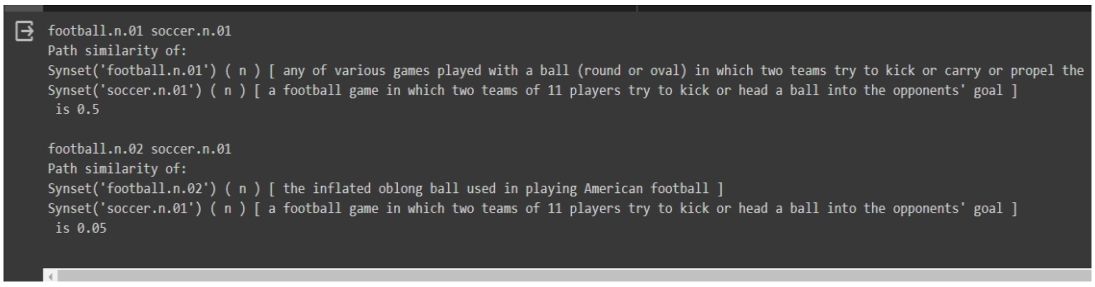
```
[Synset('active_agent.n.01'), Synset('active_voice.n.01'), Synset('active.n.03'), Synset('active.a.01'), Synset('active.s.02'), Synse
[Lemma('inactive.a.02.inactive')]
```

D. Compare two nouns.

Code: -

```
import nltk
from nltk.corpus import wordnet
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(s1.name(), s2.name())
    print("Path similarity of: ")
    print(s1, '(', s1.pos(), ')', '[', s1.definition(), ']')
    print(s2, '(', s2.pos(), ')', '[', s2.definition(), ']')
    print(" is", s1.path_similarity(s2))
    print()
```

Output: -



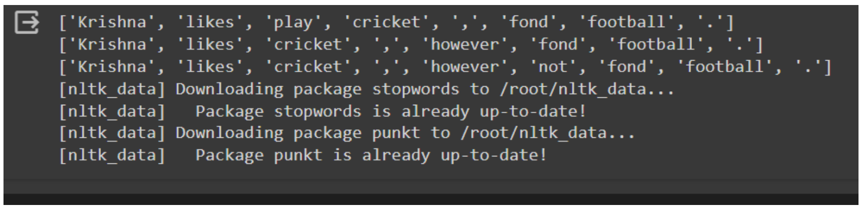
```
football.n.01 soccer.n.01
Path similarity of:
Synset('football.n.01') ( n ) [ any of various games played with a ball (round or oval) in which two teams try to kick or carry or propel the
Synset('soccer.n.01') ( n ) [ a football game in which two teams of 11 players try to kick or head a ball into the opponents' goal ]
 is 0.5

football.n.02 soccer.n.01
Path similarity of:
Synset('football.n.02') ( n ) [ the inflated oblong ball used in playing American football ]
Synset('soccer.n.01') ( n ) [ a football game in which two teams of 11 players try to kick or head a ball into the opponents' goal ]
 is 0.05
```

E. Handling stopword.

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

Code: -
```
import nltk
from nltk.corpus import stopwords
nltk.download('stopwords')
nltk.download('punkt')
from nltk.tokenize import word_tokenize
text = "Krishna likes to play cricket, however he is not too fond of
football."
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)
```

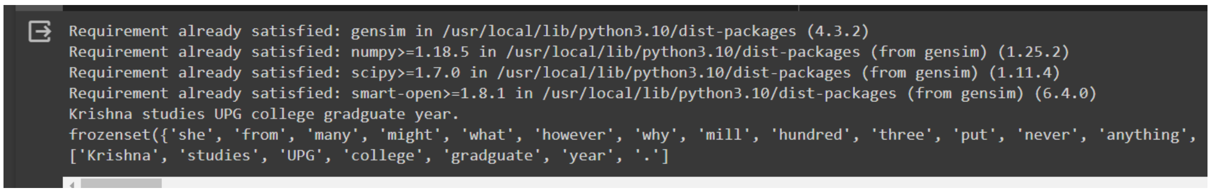
Output: -


```
['Krishna', 'likes', 'play', 'cricket', ',', 'fond', 'football', '.']
['Krishna', 'likes', 'cricket', ',', 'however', 'fond', 'football', '.']
['Krishna', 'likes', 'cricket', ',', 'however', 'not', 'fond', 'football', '.']
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
```

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

Code: -
```
!pip install gensim
import gensim
from gensim.parsing.preprocessing import remove_stopwords
text = "Krishna studies in UPG college and he will gradguate this year."
filtered_sentence = remove_stopwords(text)
print(filtered_sentence)
all_stopwords = gensim.parsing.preprocessing.STOPWORDS
print(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']))
text = "Krishna studies in UPG college and he will gradguate this year."
text_tokens = word_tokenize(text)
tokens_without_sw = [word for word in text_tokens if not word in
all_stopwords_gensim]
print(tokens_without_sw)
```

Output: -



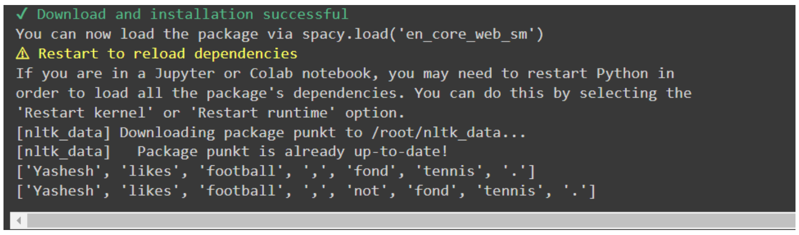
```
Requirement already satisfied: gensim in /usr/local/lib/python3.10/dist-packages (4.3.2)
Requirement already satisfied: numpy>=1.18.5 in /usr/local/lib/python3.10/dist-packages (from gensim) (1.25.2)
Requirement already satisfied: scipy>=1.7.0 in /usr/local/lib/python3.10/dist-packages (from gensim) (1.11.4)
Requirement already satisfied: smart-open>=1.8.1 in /usr/local/lib/python3.10/dist-packages (from gensim) (6.4.0)
Krishna studies UPG college gradguate year.
frozenset({'she', 'from', 'many', 'might', 'what', 'however', 'why', 'mill', 'hundred', 'three', 'put', 'never', 'anything',
['Krishna', 'studies', 'UPG', 'college', 'gradguate', 'year', '.']
```

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

Code: -

```
!pip install spacy
!python -m spacy download en_core_web_sm
!python -m spacy download en
import spacy
import nltk
nltk.download('punkt')
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: -



```
✓ Download and installation successful
You can now load the package via spacy.load('en_core_web_sm')
⚠ Restart to reload dependencies
If you are in a Jupyter or Colab notebook, you may need to restart Python in
order to load all the package's dependencies. You can do this by selecting the
'Restart kernel' or 'Restart runtime' option.
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
['Yashesh', 'likes', 'football', ',', 'fond', 'tennis', '.']
['Yashesh', 'likes', 'football', ',', 'not', 'fond', 'tennis', '.']
```

Practical 4

Theory:

Tokenization breaks text into smaller parts for easier machine analysis, helping machines understand human language. Tokenization, in the realm of Natural Language Processing (NLP) and machine learning, refers to the process of converting a sequence of text into smaller parts, known as tokens.

The split() method is a built-in function in Python that splits a string into a list of substrings based on a specified separator.

Regular expressions allow for more advanced tokenization by defining patterns for word boundaries, punctuation, etc.

NLTK (Natural Language Toolkit) provides a powerful toolkit for text processing tasks, including tokenization.

SpaCy is a modern NLP library that provides efficient tokenization and other NLP functionalities.

Keras, a deep learning library, provides text preprocessing utilities including tokenization.

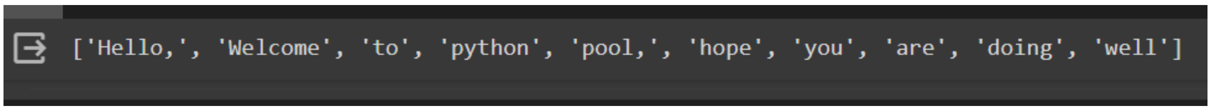
Gensim, a topic modeling library, offers text preprocessing functionalities, including tokenization.

A. Tokenization using Python's split() function

Code: -

```
sent = "Hello, Welcome to python pool, hope you are doing well"
tokens = sent.split()
print(tokens)
```

Output: -



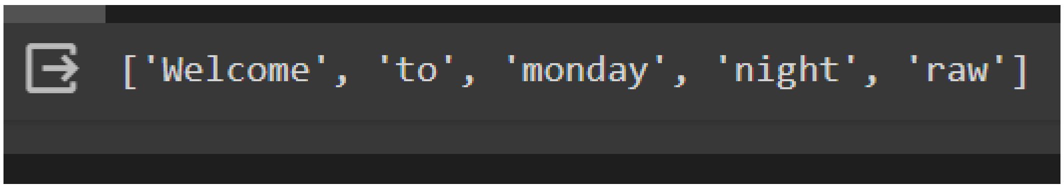
```
['Hello,', 'Welcome', 'to', 'python', 'pool,', 'hope', 'you', 'are', 'doing', 'well']
```

B. Tokenization using Regular Expressions (RegEx)

Code: -

```
import re
string = "Welcome! to monday night raw."
token = re.findall(r'\w+',string)
print(token)
```

Output: -



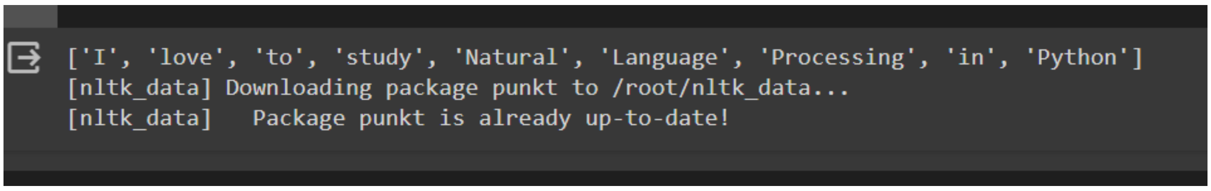
```
['Welcome', 'to', 'monday', 'night', 'raw']
```

C. Tokenization using NLTK

Code: -

```
import nltk
nltk.download('punkt')
from nltk.tokenize import word_tokenize
# Create a string input
str = "I love to study Natural Language Processing in Python"
# Use tokenize method
print(word_tokenize(str))
```

Output: -



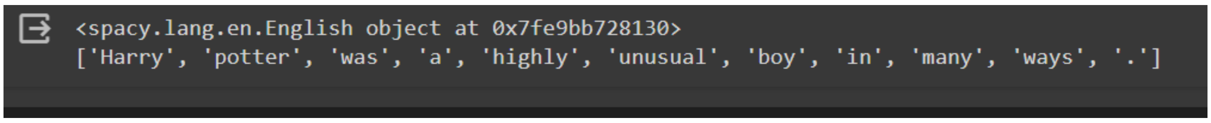
```
['I', 'love', 'to', 'study', 'Natural', 'Language', 'Processing', 'in', 'Python']
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
```

D. Tokenization using the spaCy library

Code: -

```
import spacy
nlp = spacy.blank("en")
print(nlp)
# Create a string input
str = "Harry potter was a highly unusual boy in many ways."
# Create an instance of document;
# doc object is a container for a sequence of Token objects.
doc = nlp(str)
# Read the words; Print the words
words = [word.text for word in doc]
print(words)
```

Output: -



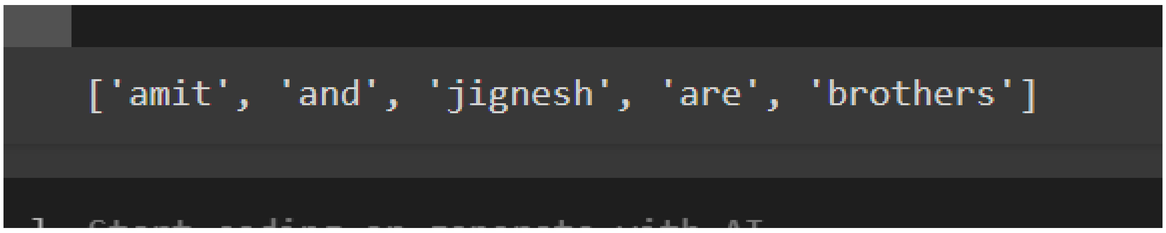
```
<spacy.lang.en.English object at 0x7fe9bb728130>
['Harry', 'potter', 'was', 'a', 'highly', 'unusual', 'boy', 'in', 'many', 'ways', '.']
```

E. Tokenization using Keras

Code: -

```
from keras.preprocessing.text import text_to_word_sequence
# define the document
text = 'Amit and Jignesh are brothers.'
# tokenize the document
result = text_to_word_sequence(text)
print(result)
```

Output: -



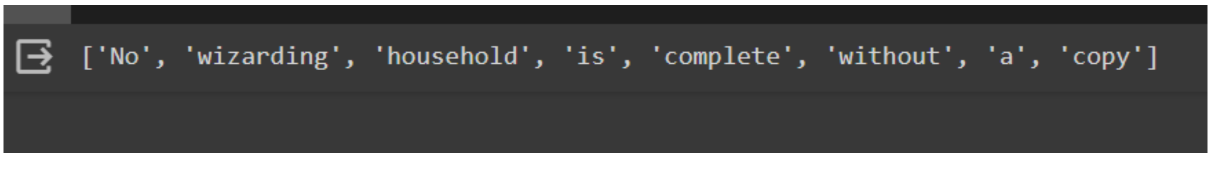
```
['amit', 'and', 'jignesh', 'are', 'brothers']
```

F. Tokenization using Gensim

Code: -

```
from gensim.utils import tokenize
# Create a string input
str = "No wizarding household is complete without a copy."
# tokenizing the text
list(tokenize(str))
```

Output: -



```
['No', 'wizarding', 'household', 'is', 'complete', 'without', 'a', 'copy']
```

Practical 5

Aim: Illustrate part of speech tagging.

Theory:

Part-of-speech tagging involves assigning a specific grammatical category (tag) to each word in a sentence, such as noun, verb, adjective, etc. This process helps in understanding the syntactic structure of the text and is crucial for various NLP tasks.

Chunking, also known as shallow parsing, involves grouping adjacent words in a sentence into syntactically related phrases, such as noun phrases (NP), verb phrases (VP), etc. This process helps in identifying higher-level structures and extracting meaningful information from text.

Named Entity Recognition (NER) is a natural language processing task that involves identifying and categorizing named entities (such as persons, organizations, locations, dates, etc.) mentioned in unstructured text. NER is crucial for various NLP applications such as information extraction, question answering, and document summarization. NER algorithms analyze the sequence of tokens and their corresponding POS tags to identify spans of text that represent named entities. This is typically done using machine learning models such as Conditional Random Fields (CRF), Bidirectional LSTM (BiLSTM), or Transformer-based models.

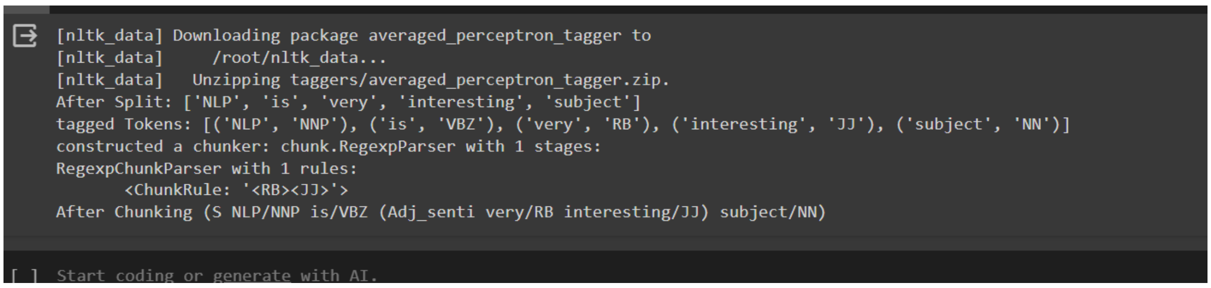
Performing Named Entity Recognition (NER) using NLTK's Treebank corpus involves several steps, including tokenization, POS tagging, and chunking. It's not possible to directly create a visualization of the entire process using NLTK's Treebank corpus alone. The Treebank corpus provides labeled sentences with POS tags, but it doesn't directly provide information about named entities. While NLTK's Treebank corpus provides labeled sentences with POS tags, for a complete NER process, you may need additional resources such as labeled named entity data or a custom NER model trained on such data.

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

Code: -

```
import nltk
from nltk import pos_tag
from nltk import RegexpParser
nltk.download('averaged_perceptron_tagger')
text ="NLP is very interesting subject".split()
print("After Split:",text)
tokens_tag = pos_tag(text)
print("tagged Tokens:",tokens_tag)
patterns= """Adj_senti :{<RB><JJ>} """
chunker = RegexpParser(patterns)
print("constructed a chunker:",chunker)
output = chunker.parse(tokens_tag)
print("After Chunking",output)
```

Output: -



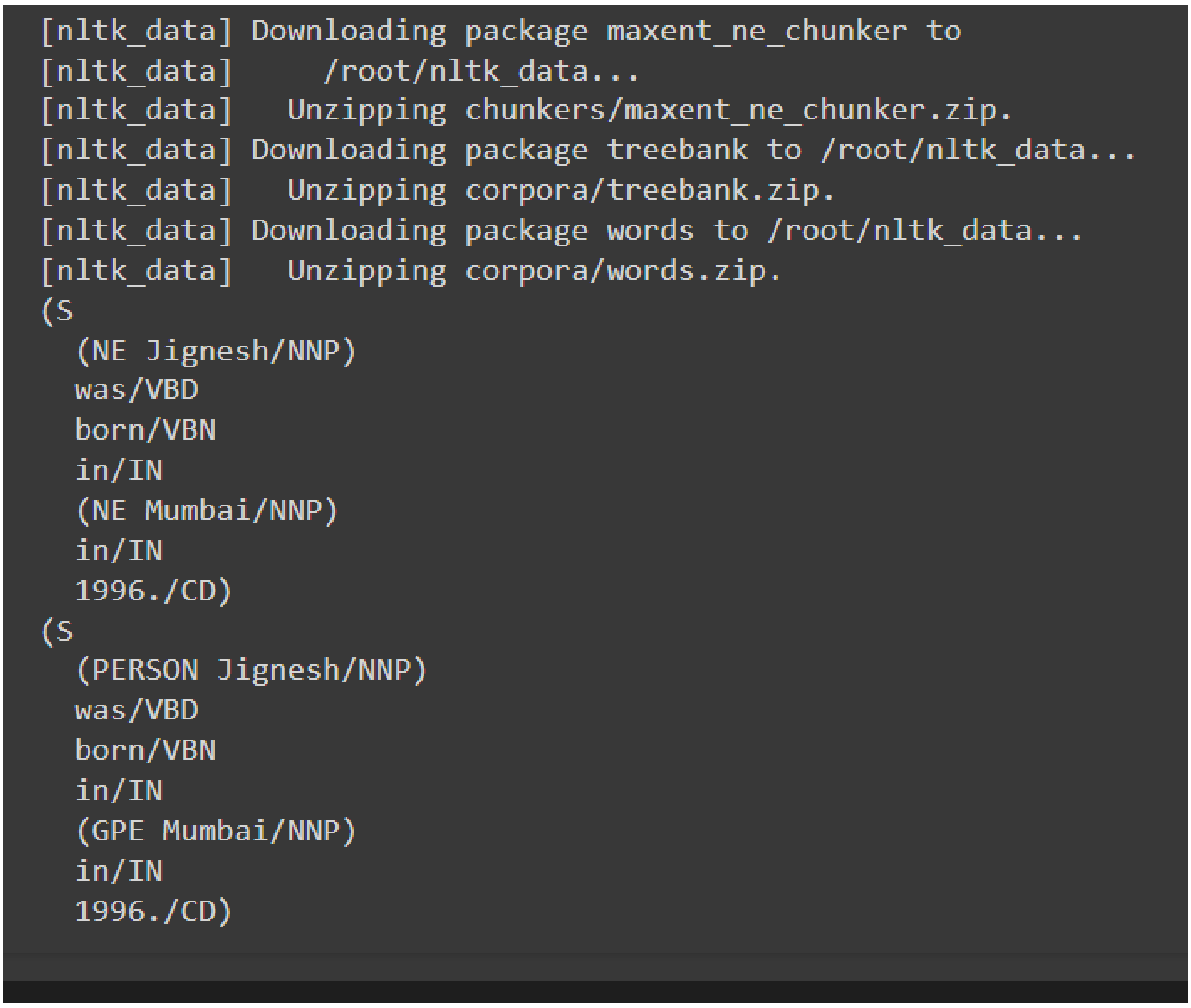
```
[nltk_data] Downloading package averaged_perceptron_tagger to
[nltk_data]     /root/nltk_data...
[nltk_data]   Unzipping taggers/averaged_perceptron_tagger.zip.
After Split: ['NLP', 'is', 'very', 'interesting', 'subject']
tagged Tokens: [('NLP', 'NNP'), ('is', 'VBZ'), ('very', 'RB'), ('interesting', 'JJ'), ('subject', 'NN')]
constructed a chunker: chunk.RegexpParser with 1 stages:
RegexpChunkParser with 1 rules:
       <ChunkRule: '<RB><JJ>'>
After Chunking (S NLP/NNP is/VBZ (Adj_senti very/RB interesting/JJ) subject/NN)
```

B. Named Entity recognition of user defined text.

Code: -

```
!python -m nltk.downloader averaged_perceptron_tagger
import nltk
nltk.download('maxent_ne_chunker')
nltk.download('treebank')
nltk.download('words')
from nltk import pos_tag
text="Jignesh was born in Mumbai in 1996."
text_tag=pos_tag(text.split())
sent = nltk.corpus.treebank.tagged_sents()[1]
print(nltk.ne_chunk(text_tag, binary=True))
print(nltk.ne_chunk(text_tag))
```

Output: -



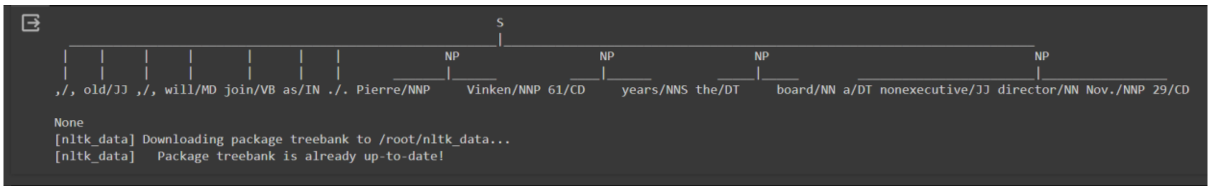
```
[nltk_data] Downloading package maxent_ne_chunker to
[nltk_data]     /root/nltk_data...
[nltk_data]   Unzipping chunkers/maxent_ne_chunker.zip.
[nltk_data] Downloading package treebank to /root/nltk_data...
[nltk_data]   Unzipping corpora/treebank.zip.
[nltk_data] Downloading package words to /root/nltk_data...
[nltk_data]   Unzipping corpora/words.zip.
(S
  (NE Jignesh/NNP)
  was/VBD
  born/VBN
  in/IN
  (NE Mumbai/NNP)
  in/IN
  1996./CD)
(S
  (PERSON Jignesh/NNP)
  was/VBD
  born/VBN
  in/IN
  (GPE Mumbai/NNP)
  in/IN
  1996./CD)
```

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

Code: -

```
import nltk
nltk.download('treebank')
from nltk.corpus import treebank_chunk
sentence = treebank_chunk.chunked_sents()[0]

# Pretty printing the parse tree
print(sentence.pretty_print())
```

Output: -



```
                                                           S
 __________________________________________________________|_____________________________________________________________
 |    |     |      |       |       |    |             NP                  NP                  NP                          NP
 |    |     |      |       |       |    |       ______|_____         _____|_____        ______|_____      _____________________|_______________
,/, old/JJ ,/, will/MD join/VB as/IN ./. Pierre/NNP     Vinken/NNP 61/CD     years/NNS the/DT     board/NN a/DT nonexecutive/JJ director/NN Nov./NNP 29/CD

None
[nltk_data] Downloading package treebank to /root/nltk_data...
[nltk_data]   Package treebank is already up-to-date!
```

Practical 6

Theory:

In NLTK, a grammar is typically defined using a formalism called a Context-Free Grammar (CFG). A CFG consists of a set of production rules that describe how to generate valid sentences in a language. Each rule specifies how to rewrite a non-terminal symbol (a category) into a sequence of terminal symbols (words) or other non-terminal symbols.

In this example, we define a grammar that describes simple English sentences consisting of nouns (N), verbs (V), determiners (Det), prepositions (P), and prepositional phrases (PP). Each production rule specifies how to expand a non-terminal symbol into a sequence of terminal symbols or other non-terminal symbols.

Once the grammar is defined, we create a parser using nltk.ChartParser, which allows us to parse sentences according to the defined grammar. We then parse a sample sentence using the parser and print the resulting parse trees.

This is just a basic example of defining and using a CFG grammar in NLTK. CFGs can be much more complex and can describe the syntax of various languages in detail. They are widely used in NLP for tasks such as syntactic parsing, grammar checking, and text generation.

To accept an input string with a regular expression using finite automata (FA), we need to design a finite automaton that recognizes strings matching the given regular expression. In this case, the regular expression is "101+" which means it accepts strings containing one or more occurrences of the substring "101". This finite automaton will accept strings that contain one or more occurrences of the substring "101".

To accept input strings with the regular expression (a+b)*bba, we need to design a finite automaton (FA) that recognizes strings following this pattern. This regular expression describes strings that start with any combination of 'a' and 'b' (including no symbols), followed by the substring 'bba' at the end. This finite automaton will accept strings that start with any combination of 'a' and 'b' and end with the substring 'bba'.

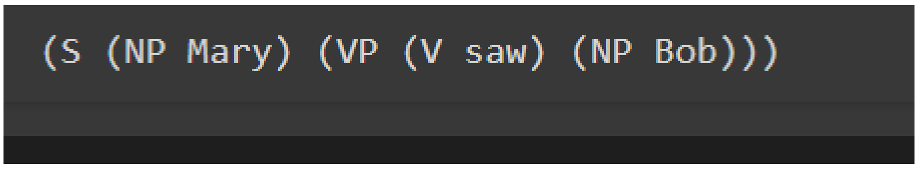
Deductive chart parsing is a parsing technique used in natural language processing (NLP) to parse sentences based on a given context-free grammar (CFG). The process involves building a parse chart that represents all possible parses of the input sentence according to the grammar rules. We parse the input sentence using the chart parser, which generates parse trees representing possible parses of the input sentence.

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

Code:

```
import nltk
from nltk import CFG
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)
```

Output:



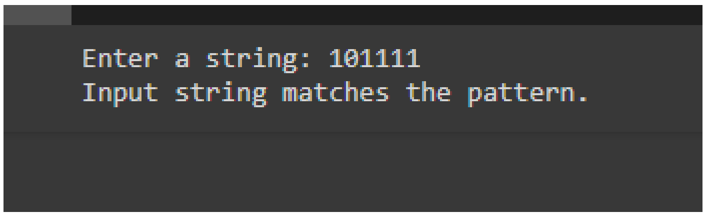
```
(S (NP Mary) (VP (V saw) (NP Bob)))
```

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

Code:

```
import re
# Define the regular expression pattern
pattern = r"101+"
# Accept input string
input_string = input("Enter a string: ")
# Check if the input string matches the pattern
if re.fullmatch(pattern, input_string):
 print("Input string matches the pattern.")
else:
 print("Input string does not match the pattern.")
```

Output:



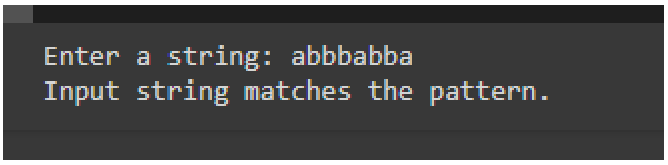
```
Enter a string: 101111
Input string matches the pattern.
```

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

Code:

```
import re
# Define the regular expression pattern
pattern = r"^(a+b)*bba"
# Accept input string
input_string = input("Enter a string: ")
# Check if the input string matches the pattern
if re.match(pattern, input_string):
 print("Input string matches the pattern.")
else:
 print("Input string does not match the pattern.")
```

Output:



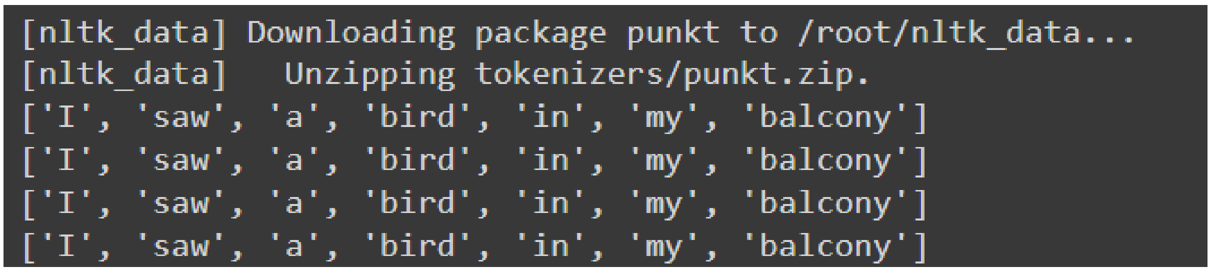
```
Enter a string: abbbabba
Input string matches the pattern.
```

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

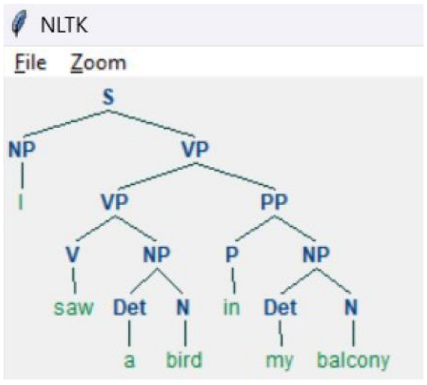
Code: -

```
import nltk
nltk.download('punkt')
from nltk import tokenize
grammar1 = nltk.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'
""")
sentence = "I saw a bird in my balcony"
for index in range(len(sentence)):
  all_tokens = tokenize.word_tokenize(sentence)
  print(all_tokens)
# all_tokens = ['I', 'saw', 'a', 'bird', 'in', 'my', 'balcony']
  parser = nltk.ChartParser(grammar1)
for tree in parser.parse(all_tokens):
  print(tree)
  tree.draw()
```

Output:



```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt.zip.
['I', 'saw', 'a', 'bird', 'in', 'my', 'balcony']
['I', 'saw', 'a', 'bird', 'in', 'my', 'balcony']
['I', 'saw', 'a', 'bird', 'in', 'my', 'balcony']
['I', 'saw', 'a', 'bird', 'in', 'my', 'balcony']
```



Practical 7

Aim. Study PorterStemmer, LancasterStemmer, RegexpStemmer, SnowballStemmer. Study WordNetLemmatizer.

Theory:

In the realm of natural language processing (NLP), stemming and lemmatization are crucial techniques for reducing words to their root forms, thereby improving text analysis and information retrieval. NLTK provides several stemmers and lemmatizers, each with its own characteristics and applications.

1. PorterStemmer: The Porter stemming algorithm, developed by Martin Porter, is one of the most widely used stemming algorithms. It applies a set of heuristic rules to remove suffixes from words, aiming to produce the stem or root form of the word. While simple and computationally efficient, the PorterStemmer may not always generate linguistically valid stems, as it focuses primarily on suffix removal without considering linguistic context.

2. LancasterStemmer: The Lancaster stemming algorithm, developed by Chris D. Paice, is another popular stemming algorithm known for its aggressive stemming behavior. It applies more aggressive rules compared to the Porter algorithm, resulting in shorter stems. However, this aggressiveness may lead to over-stemming, where the stems produced may not always be valid English words.

3. RegexpStemmer: The RegexpStemmer allows users to define custom stemming rules based on regular expressions. This flexibility makes it suitable for handling specific cases or languages where standard stemming algorithms may not be effective. Users can define patterns to match and remove prefixes or suffixes from words according to their specific requirements.

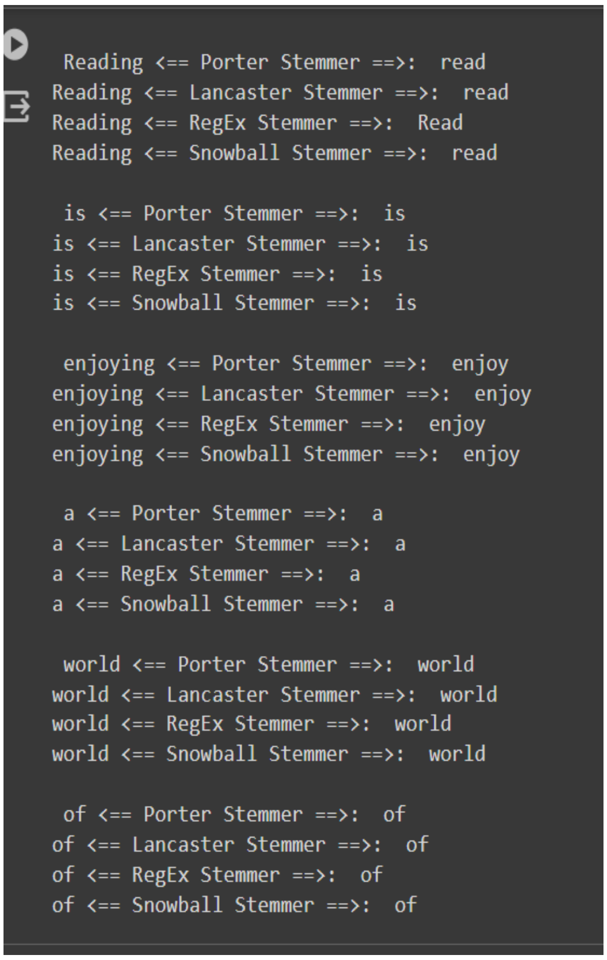
4. SnowballStemmer: The Snowball stemming algorithm, also known as the Porter2 stemming algorithm, is an improved version of the original Porter algorithm. It offers better performance and accuracy by addressing some of the limitations of the Porter algorithm. SnowballStemmer supports stemming for multiple languages and provides more control over stemming behavior through customizable language-specific algorithms.

5. WordNetLemmatizer: Unlike stemmers that simply remove suffixes to produce root forms, the WordNetLemmatizer in NLTK utilizes WordNet, a lexical database of English words, to perform lemmatization. Lemmatization aims to identify the base or dictionary form of a word, known as the lemma, by considering its morphological variants and linguistic context. The WordNetLemmatizer maps words to their corresponding lemmas based on WordNet's lexical hierarchy and morphological information, resulting in linguistically valid lemmatization.

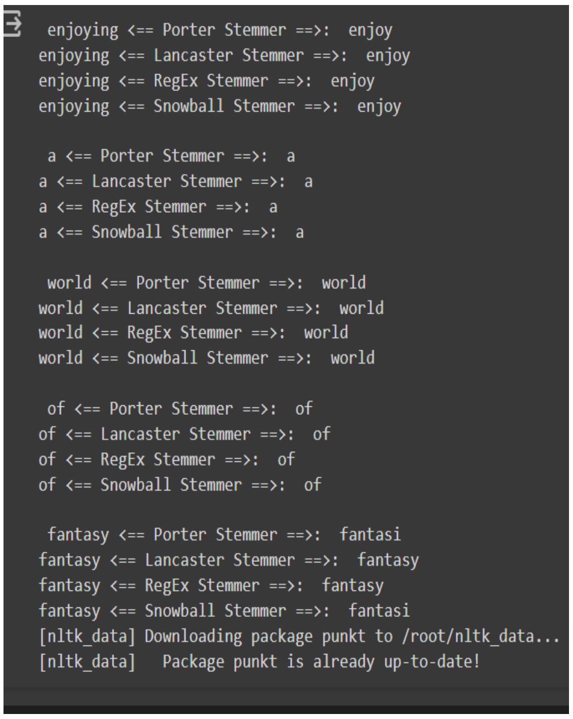
Code:

```
import nltk
from nltk.stem import PorterStemmer
from nltk.stem import LancasterStemmer
from nltk.stem import RegexpStemmer
from nltk.stem.snowball import SnowballStemmer

from nltk.tokenize import word_tokenize
nltk.download('punkt')
ps = PorterStemmer()
ls = LancasterStemmer()
rs = RegexpStemmer('ing$|s$|e$|able$', min=4)
ss = SnowballStemmer(language='english')
# choose some words to be stemmed
sentence = "Reading is enjoying a world of fantasy"
words = word_tokenize(sentence)

for w in words:
    print("\n",w, "<== Porter Stemmer ==>: ", ps.stem(w))
    print(w, "<== Lancaster Stemmer ==>: ", ls.stem(w))
    print(w, "<== RegEx Stemmer ==>: ", rs.stem(w))
    print(w, "<== Snowball Stemmer ==>: ", ss.stem(w))
```

Output:



```
 Reading <== Porter Stemmer ==>:  read
Reading <== Lancaster Stemmer ==>:  read
Reading <== RegEx Stemmer ==>:  Read
Reading <== Snowball Stemmer ==>:  read

 is <== Porter Stemmer ==>:  is
is <== Lancaster Stemmer ==>:  is
is <== RegEx Stemmer ==>:  is
is <== Snowball Stemmer ==>:  is

 enjoying <== Porter Stemmer ==>:  enjoy
enjoying <== Lancaster Stemmer ==>:  enjoy
enjoying <== RegEx Stemmer ==>:  enjoy
enjoying <== Snowball Stemmer ==>:  enjoy

 a <== Porter Stemmer ==>:  a
a <== Lancaster Stemmer ==>:  a
a <== RegEx Stemmer ==>:  a
a <== Snowball Stemmer ==>:  a

 world <== Porter Stemmer ==>:  world
world <== Lancaster Stemmer ==>:  world
world <== RegEx Stemmer ==>:  world
world <== Snowball Stemmer ==>:  world

 of <== Porter Stemmer ==>:  of
of <== Lancaster Stemmer ==>:  of
of <== RegEx Stemmer ==>:  of
of <== Snowball Stemmer ==>:  of
```



```
 enjoying <== Porter Stemmer ==>:  enjoy
enjoying <== Lancaster Stemmer ==>:  enjoy
enjoying <== RegEx Stemmer ==>:  enjoy
enjoying <== Snowball Stemmer ==>:  enjoy

 a <== Porter Stemmer ==>:  a
a <== Lancaster Stemmer ==>:  a
a <== RegEx Stemmer ==>:  a
a <== Snowball Stemmer ==>:  a

 world <== Porter Stemmer ==>:  world
world <== Lancaster Stemmer ==>:  world
world <== RegEx Stemmer ==>:  world
world <== Snowball Stemmer ==>:  world

 of <== Porter Stemmer ==>:  of
of <== Lancaster Stemmer ==>:  of
of <== RegEx Stemmer ==>:  of
of <== Snowball Stemmer ==>:  of

 fantasy <== Porter Stemmer ==>:  fantasi
fantasy <== Lancaster Stemmer ==>:  fantasy
fantasy <== RegEx Stemmer ==>:  fantasy
fantasy <== Snowball Stemmer ==>:  fantasi
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
```

Practical 8

Aim: Implement Naive Bayes classifier

Theory:

The Naive Bayes classifier is a probabilistic machine learning algorithm used for classification tasks. It is based on Bayes' theorem and assumes that features are conditionally independent given the class label, which is a "naive" assumption but often works well in practice, especially with text classification tasks.

Naive Bayes algorithm is used for classification problems. It is highly used in text classification. In text classification tasks, data contains high dimension (as each word represent one feature in the data). It is used in spam filtering, sentiment detection, rating classification etc. The advantage of using naïve Bayes is its speed. It is fast and making prediction is easy with high dimension of data.

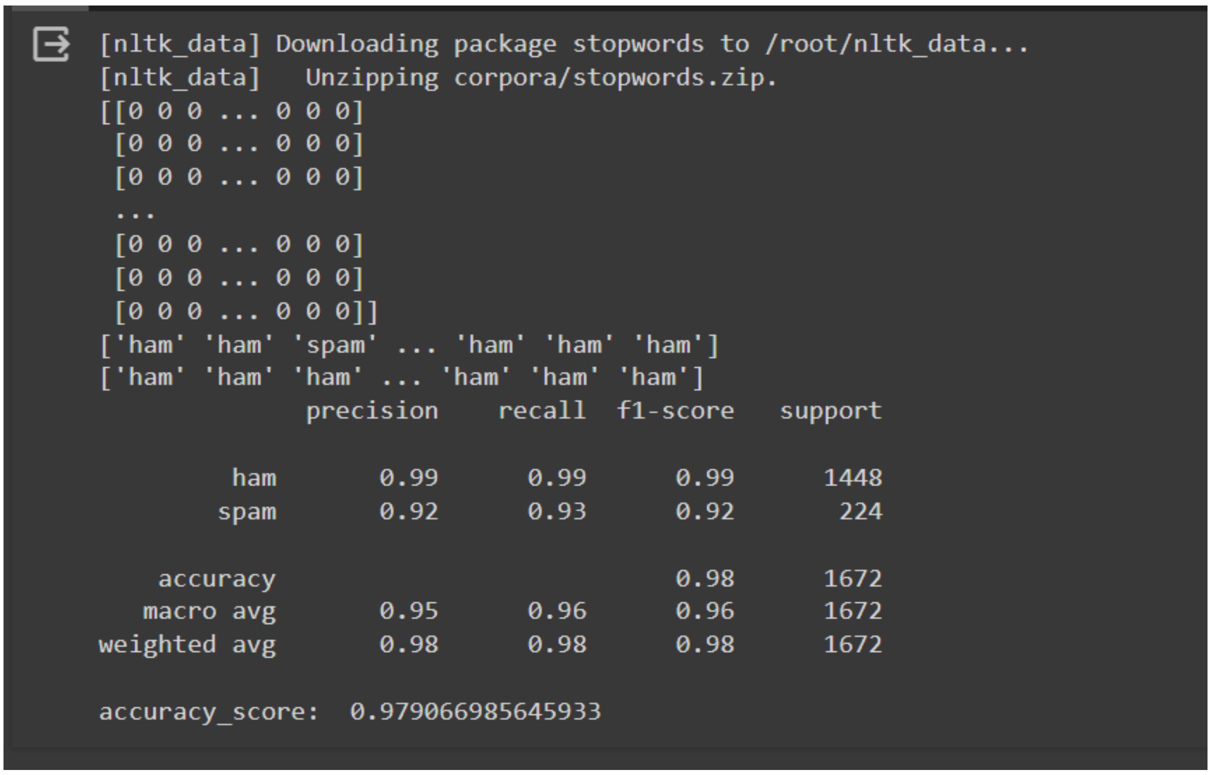
This model predicts the probability of an instance belongs to a class with a given set of feature value. It is a probabilistic classifier. It is because it assumes that one feature in the model is independent of existence of another feature. In other words, each feature contributes to the predictions with no relation between each other. In real world, this condition satisfies rarely. It uses Bayes theorem in the algorithm for training and prediction

The Naive Bayes classifier assumes that the features (or attributes) are conditionally independent given the class label. Mathematically, this can be Expressed as:

$P(x1,x2,...,xn|y)=P(x1|y)\cdot P(x2|y)\cdot\ldots\cdot P(xn|y)P(x1,x2,...,xn|y)=P(x1|y)\cdot P(x2|y)\cdot\ldots\cdot P(xn|y)$ where $x1,x2,...,xnx1,x2,...,xn$ are the features, and yy is the class label. The Naive Bayes classifier is simple, fast, and often works well in practice, especially for text classification tasks such as spam detection, sentiment analysis, and document categorization.

Code:
```
import pandas as pd
import numpy as np

sms_data = pd.read_csv("spam.csv", encoding='latin-1')

import nltk
nltk.download('stopwords')
import re
from nltk.corpus import stopwords
from nltk.stem.porter import PorterStemmer

stemming = PorterStemmer()
corpus = []

for i in range(0, len(sms_data)):
    s1 = re.sub('[^a-zA-Z]', repl=' ', string=sms_data['v2'][i])
    s1.lower()
    s1 = s1.split()
    s1 = [stemming.stem(word) for word in s1 if word not in
set(stopwords.words('english'))]
    s1 = ' '.join(s1)
    corpus.append(s1)

from sklearn.feature_extraction.text import CountVectorizer

countvectorizer = CountVectorizer()
x = countvectorizer.fit_transform(corpus).toarray()
print(x)

y = sms_data['v1'].values
print(y)

from sklearn.model_selection import train_test_split

x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.3,
stratify=y, random_state=2)

# Multinomial Naïve Bayes.
from sklearn.naive_bayes import MultinomialNB

multinomialnb = MultinomialNB()
multinomialnb.fit(x_train, y_train)

# Predicting on test data:
y_pred = multinomialnb.predict(x_test)
print(y_pred)

# Result
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.metrics import accuracy_score

print(classification_report(y_test, y_pred))
print("accuracy_score: ", accuracy_score(y_test, y_pred))
```

Output:



```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
[[0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 ...
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]]
['ham' 'ham' 'spam' ... 'ham' 'ham' 'ham']
['ham' 'ham' 'ham' ... 'ham' 'ham' 'ham']
              precision    recall  f1-score   support

         ham       0.99      0.99      0.99      1448
        spam       0.92      0.93      0.92       224

    accuracy                           0.98      1672
   macro avg       0.95      0.96      0.96      1672
weighted avg       0.98      0.98      0.98      1672

accuracy_score:  0.979066985645933
```

Practical 9

Theory:

Statistical parsing is a parsing technique used in natural language processing (NLP) that employs statistical models to parse sentences and determine their syntactic structure. Unlike rule-based parsing approaches that rely on handcrafted grammatical rules, statistical parsing learns parsing models from large corpora of annotated training data. Statistical parsing is a powerful approach for automatically analyzing the syntactic structure of natural language text and is widely used in various NLP tasks such as syntactic parsing, semantic parsing, machine translation, and information extraction.

Probabilistic parsing is a parsing technique used in natural language processing (NLP) that assigns probabilities to alternative syntactic analyses of a sentence. Instead of producing a single parse tree for a given sentence, probabilistic parsing assigns probabilities to different syntactic structures based on the likelihood of each structure being correct given the observed data. Probabilistic parsing techniques are widely used in NLP tasks such as syntactic parsing, semantic parsing, machine translation, and information extraction.

MaltParser is a system for dependency parsing of natural language text. Dependency parsing is a method used in natural language processing to analyze the grammatical structure of sentences by establishing relationships between words based on their syntactic dependencies. MaltParser uses a data-driven approach to dependency parsing, where it learns parsing models from annotated training data. It builds parsing models based on features extracted from the input sentence.

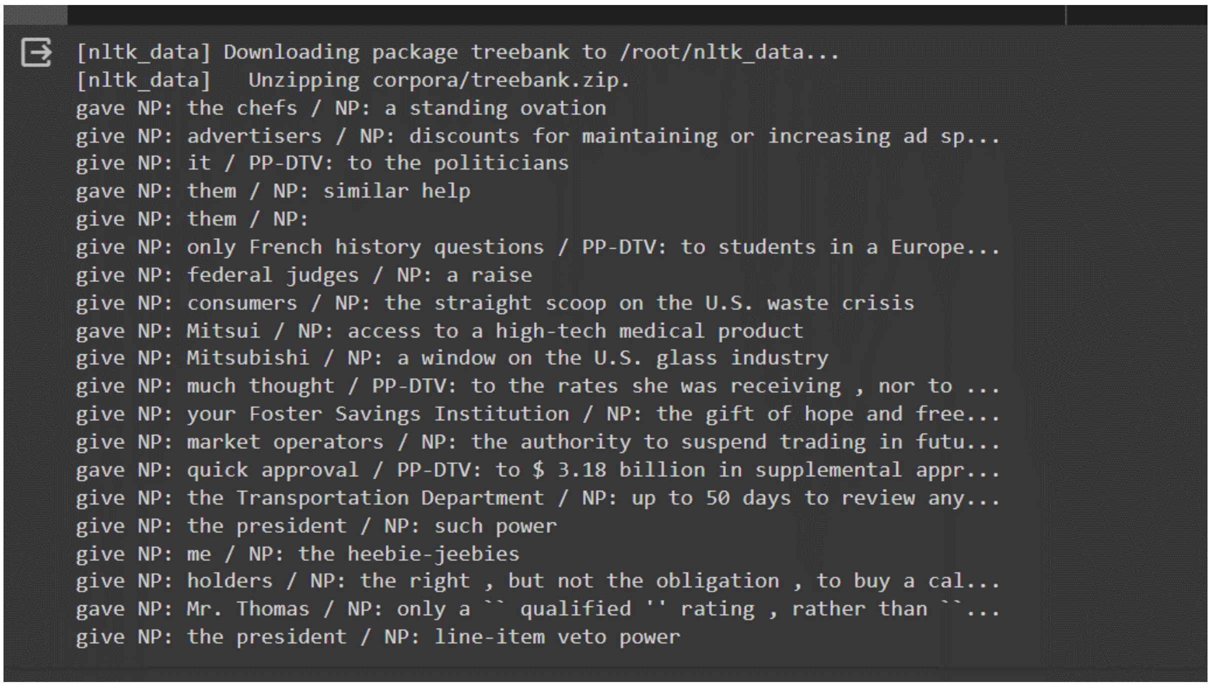
Statistical parsing:

A. Usage of Give and Gave in the Penn Treebank sample.

Code: -

```
!pip install nltk
import nltk
nltk.download('treebank')
import nltk.parse.viterbi
import nltk.parse.pchart

def give(t):
    return t.label() == 'VP' and len(t) > 2 and t[1].label() == 'NP' \
        and (t[2].label() == 'PP-DTV' or t[2].label() == 'NP') \
        and ('give' in t[0].leaves() or 'gave' in t[0].leaves())

def sent(t):
    return ' '.join(token for token in t.leaves() if token[0] not in '*-0')

def print_node(t, width):
    output = "%s %s: %s / %s: %s" % \
        (sent(t[0]), t[1].label(), sent(t[1]), t[2].label(), sent(t[2]))
    if len(output) > width:
        output = output[:width] + "..."
    print(output)

for tree in nltk.corpus.treebank.parsed_sents():
    for t in tree.subtrees(give):
        print_node(t, 72)
```

Output: -



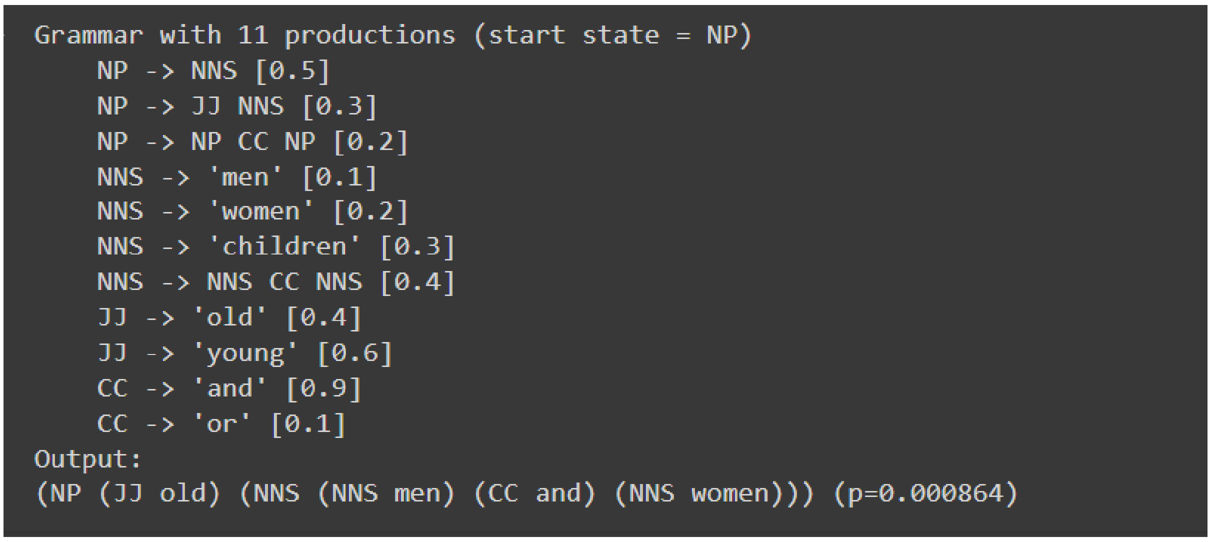
```
[nltk_data] Downloading package treebank to /root/nltk_data...
[nltk_data]   Unzipping corpora/treebank.zip.
gave NP: the chefs / NP: a standing ovation
give NP: advertisers / NP: discounts for maintaining or increasing ad sp...
give NP: it / PP-DTV: to the politicians
gave NP: them / NP: similar help
give NP: them / NP:
give NP: only French history questions / PP-DTV: to students in a Europe...
give NP: federal judges / NP: a raise
give NP: consumers / NP: the straight scoop on the U.S. waste crisis
gave NP: Mitsui / NP: access to a high-tech medical product
give NP: Mitsubishi / NP: a window on the U.S. glass industry
give NP: much thought / PP-DTV: to the rates she was receiving , nor to ...
give NP: your Foster Savings Institution / NP: the gift of hope and free...
give NP: market operators / NP: the authority to suspend trading in futu...
gave NP: quick approval / PP-DTV: to $ 3.18 billion in supplemental appr...
give NP: the Transportation Department / NP: up to 50 days to review any...
give NP: the president / NP: such power
give NP: me / NP: the heebie-jeebies
give NP: holders / NP: the right , but not the obligation , to buy a cal...
gave NP: Mr. Thomas / NP: only a `` qualified '' rating , rather than ``...
give NP: the president / NP: line-item veto power
```

B. Probabilistic parser.

Code: -

```
import nltk
from nltk import PCFG

grammar = PCFG.fromstring('''
    NP -> NNS [0.5] | JJ NNS [0.3] | NP CC NP [0.2]
    NNS -> "men" [0.1] | "women" [0.2] | "children" [0.3] | NNS CC NNS
[0.4]
    JJ -> "old" [0.4] | "young" [0.6]
    CC -> "and" [0.9] | "or" [0.1]
''')

print(grammar)

viterbi_parser = nltk.ViterbiParser(grammar)
token = "old men and women".split()
obj = viterbi_parser.parse(token)

print("Output: ")
for x in obj:
    print(x)
```

Output: -



```
Grammar with 11 productions (start state = NP)
    NP -> NNS [0.5]
    NP -> JJ NNS [0.3]
    NP -> NP CC NP [0.2]
    NNS -> 'men' [0.1]
    NNS -> 'women' [0.2]
    NNS -> 'children' [0.3]
    NNS -> NNS CC NNS [0.4]
    JJ -> 'old' [0.4]
    JJ -> 'young' [0.6]
    CC -> 'and' [0.9]
    CC -> 'or' [0.1]
Output:
(NP (JJ old) (NNS (NNS men) (CC and) (NNS women))) (p=0.000864)
```

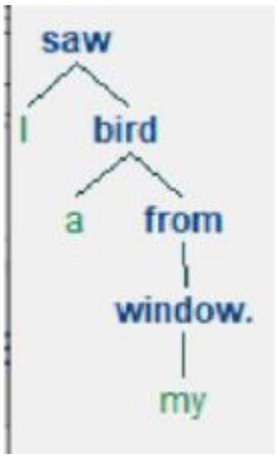
Malt parsing:

Parse a sentence and draw a tree using malt parsing

Code: -

```
!pip install nltk
import nltk
nltk.download('maltparser')
from nltk.parse.malt import MaltParser
# Load MaltParser with the pre-trained model
malt_parser = MaltParser()
# Define the sentence to parse
sentence = 'I saw a bird from my window"
# Parse the sentence
parsed_sentence = malt_parser.parse_one(sentence.split())
# Print the dependency parse tree
print(parsed_sentence.tree())
```

Output: -


saw
I
bird
a
from
window.
my