SNS COLLEGE OF ENGINEERING

An Autonomous Institution

(Accredited by NAAC-UGC with'A' Grade, Approved by AICTE & Affiliated to Anna University)

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND DATA SCIENCE

Academic Year 2024-2025

19AD603 –NATURAL LANGUAGE PROCCESSING LABORATORY

LAB MANUAL

List of Experiments

Ex No	
1	Text Preprocessing
2	Morphological Analysis
3	N-Gram Model
4	POS Tagging
5	Chunking
6	Named Entity Recognition
7	Word Generation
8	Sentiment Analysis
9	SPAM Classification
10	Autocorrect
11	NLP Applications - Chat bot Miniproject

Experiment No. 1

Text Preprocessing

Aim: To study Preprocessing of text (Tokenization, Filtration, Script Validation, Stop Word Removal)

Theory

To preprocess your text simply means to bring your text into a form that is predictable and analyzable for your task. A task here is a combination of approach and domain. Machine Learning needs data in the numeric form. We basically used encoding technique (BagOfWord, Bi-gram,n-gram, TF-IDF, Word2Vec) to encode text into numeric vector. But before encoding we first need to clean the text data and this process to prepare (or clean) text data before encoding is called text preprocessing, this is the very first step to solve the NLP problems

Tokenization:
Tokenization is about splitting strings of text into smaller pieces, or "tokens". Paragraphs can be tokenized into sentences and sentences can be tokenized into words. There are various ways to perform tokenization, using python split(), importing Regular expression, using NLTK, using spaCy library, keras and genism. Tokenization can be performed on sentence level and world level. One major drawback of using Python's split() method is that we can use only one separator at a time.
Filtration:

Similarly, if we are doing simple word counts, or trying to visualize our text with a word cloud, stopwords are some of the most frequently occurring words but don't really tell us anything. We're often better off tossing the stopwords out of the text. By checking the Filter Stop words option in the Text Pre-processing tool, you can automatically filter these words out.

Stopword Removal

Stop words are a set of commonly used words in a language. Examples of stop words in English are "a", "the", "is", "are" and etc. The intuition behind using stop words is that, by removing low information words from text, we can focus on the important words instead.

For example, in the context of a search system, if your search query is “what is text preprocessing?”, you want the search system to focus on surfacing documents that talk about text preprocessing over documents that talk about what is. This can be done by preventing all words from your stop word list from being analyzed.

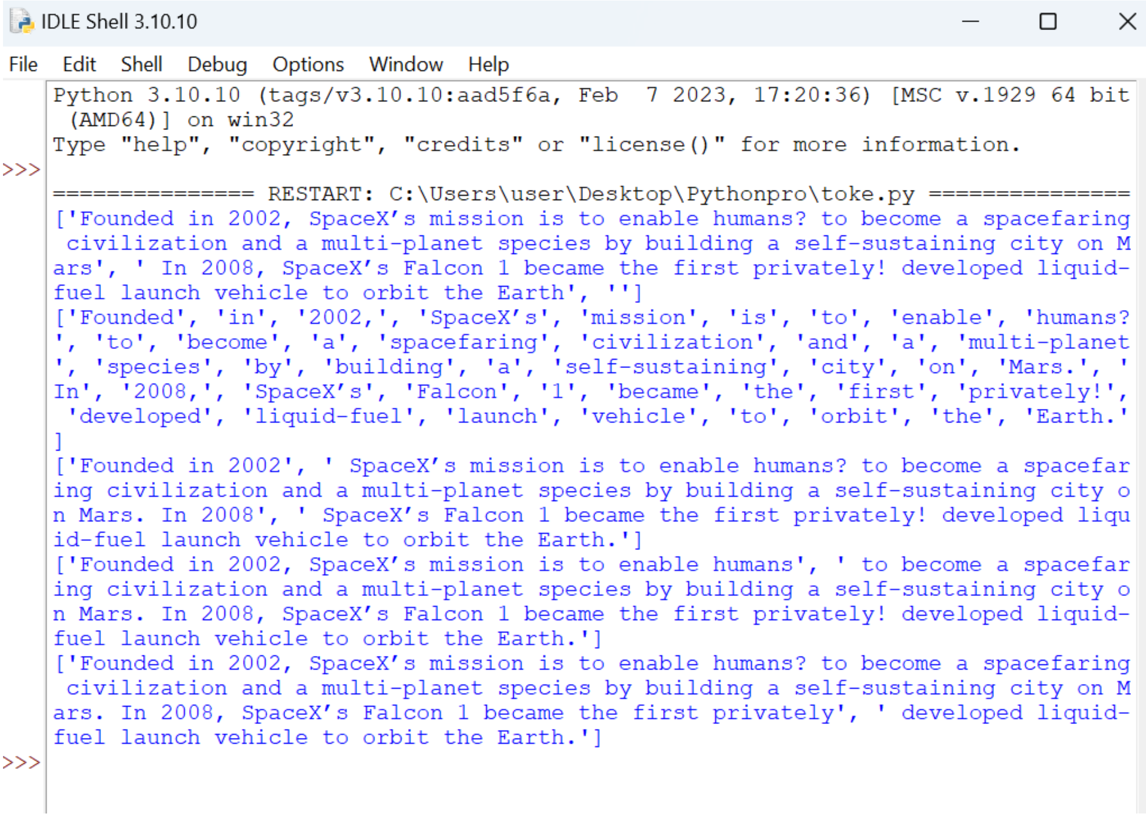
Stop words are commonly applied in search systems, text classification applications, topic modeling, topic extraction and others. In my experience, stop word removal, while effective in search and topic extraction systems, showed to be non-critical in classification systems. However, it does help reduce the number of features in consideration which helps keep your models decently sized

Tokenization Using Split function

Program

```
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."""
tokenized_text=text.split('.')
tokenized_text1=text.split()
tokenized_text2=text.split(‘,’)
tokenized_text3=text.split(‘?’)
tokenized_text4=text.split(‘!’)
print(tokenized_text)
print(tokenized_text1)
print(tokenized_text2)
print(tokenized_text3)
print(tokenized_text4)
```

output



```
IDLE Shell 3.10.10
File  Edit  Shell  Debug  Options  Window  Help
Python 3.10.10 (tags/v3.10.10:aad5f6a, Feb  7 2023, 17:20:36) [MSC v.1929 64 bit
 (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
=============== RESTART: C:\Users\user\Desktop\Pythonpro\toke.py ===============
['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 M
ars', ' In 2008, SpaceX's Falcon 1 became the first privately! developed liquid-
fuel launch vehicle to orbit the Earth', '']
['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.'
]
['Founded in 2002', ' SpaceX's mission is to enable humans? to become a spacefar
ing civilization and a multi-planet species by building a self-sustaining city o
n Mars. In 2008', ' SpaceX's Falcon 1 became the first privately! developed liqu
id-fuel launch vehicle to orbit the Earth.']
['Founded in 2002, SpaceX's mission is to enable humans', ' to become a spacefar
ing civilization and a multi-planet species by building a self-sustaining city o
n Mars. In 2008, SpaceX's Falcon 1 became the first privately! developed liquid-
fuel launch vehicle to orbit the Earth.']
['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 M
ars. In 2008, SpaceX's Falcon 1 became the first privately', ' developed liquid-
fuel launch vehicle to orbit the Earth.']
>>>
```

The above program splits the paragraph based on period, whitespaces, comma, question mark and exclamation.

Required Library

A RegEx, or Regular Expression, is a sequence of characters that forms a search pattern. RegEx can be used to check if a string contains the specified search pattern. Sentence level tokenization.

Python has a built-in package called re, which can be used to work with Regular Expressions.

Tokenization Using Regular Expression

Program

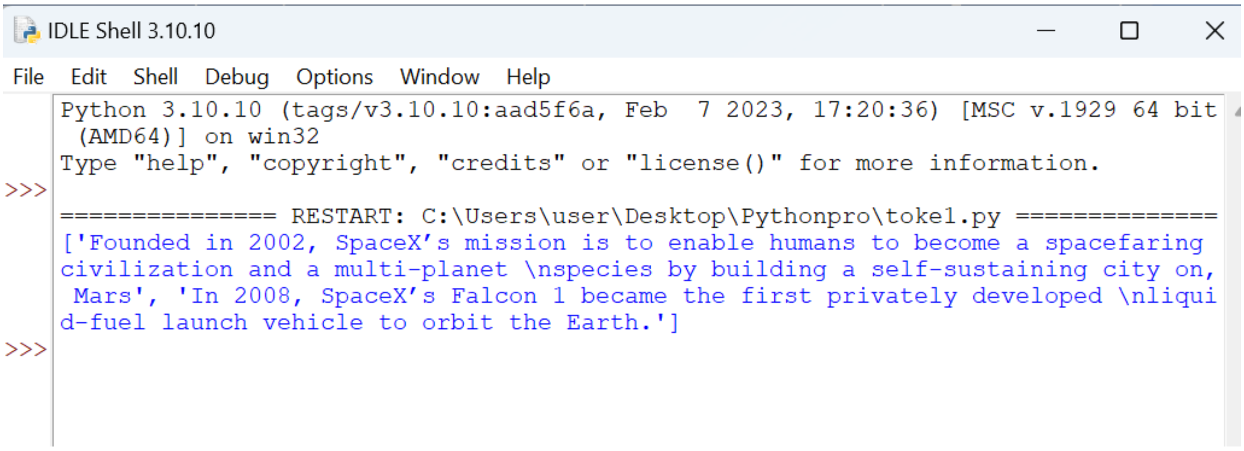
import re

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

sentences = re.compile('[.!?] ').split(text)

print(sentences)

Output



Required Library

NLTK is a standard python library with prebuilt functions and utilities for the ease of use and implementation. It is one of the most used libraries for natural language processing and computational linguistics.

Intall nltk

Pip install nltk -all

Tokenization Using NLTK

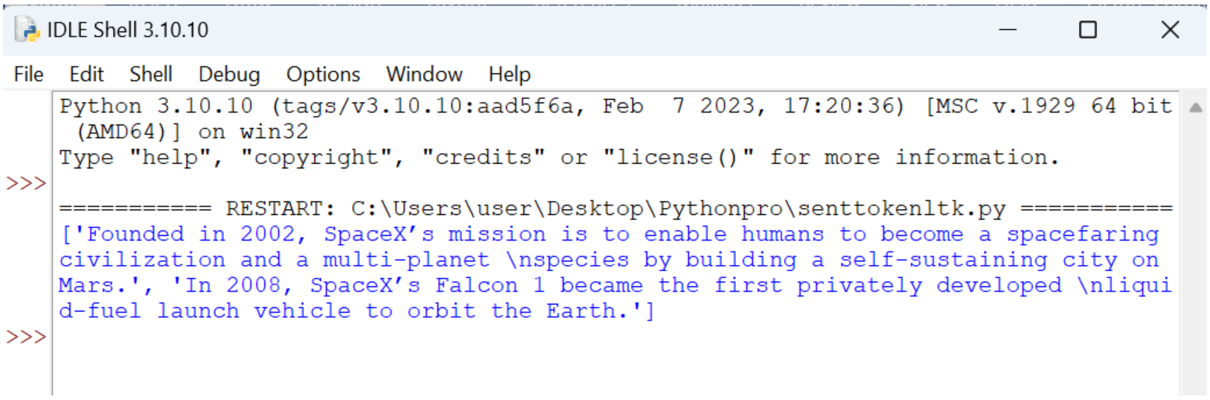
Program

from nltk.tokenize import sent_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."""

sent_tokenize(text)

Output



Word Tokenization

Tokenization Using Regular Expression

Program

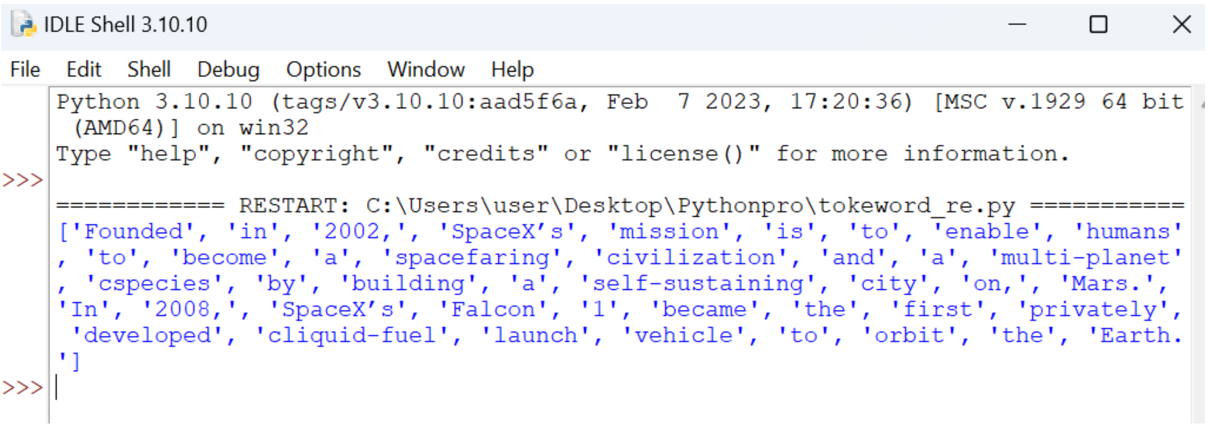
import re

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

spaces = r"\s+"

print(re.split(spaces, text))

Output



Tokenization Using NLTK

Program

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."""

print(word_tokenize(text))

Output

```
IDLE Shell 3.10.10
File  Edit  Shell  Debug  Options  Window  Help
Python 3.10.10 (tags/v3.10.10:aad5f6a, Feb  7 2023, 17:20:36) [MSC v.1929 64 bit
 (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
=========== RESTART: C:\Users\user\Desktop\Pythonpro\tokeword_nltk.py ==========
['Founded', 'in', '2002', ',', 'SpaceX', ''', 's', 'mission', 'is', 'to', 'enabl
e', 'humans', 'to', 'become', 'a', 'spacefaring', 'civilization', 'and', 'a', 'm
ulti-planet', 'species', 'by', 'building', 'a', 'self-sustaining', 'city', 'on',
 'Mars', '.', 'In', '2008', ',', 'SpaceX', ''', 's', 'Falcon', '1', 'became', 't
he', 'first', 'privately', 'developed', 'liquid-fuel', 'launch', 'vehicle', 'to'
, 'orbit', 'the', 'Earth', '.']
>>>
```

Filtering

Stopword Removal

Program

import nltk

text = "This is an example text for stopword removal and filtering. This is done using NLTK's stopwords."

words = nltk.word_tokenize(text)

print("Unfiltered: ", words)

stopwords = nltk.corpus.stopwords.words("english")

cleaned = [word for word in words if word not in stopwords]

print("Filtered: ", cleaned)

Output

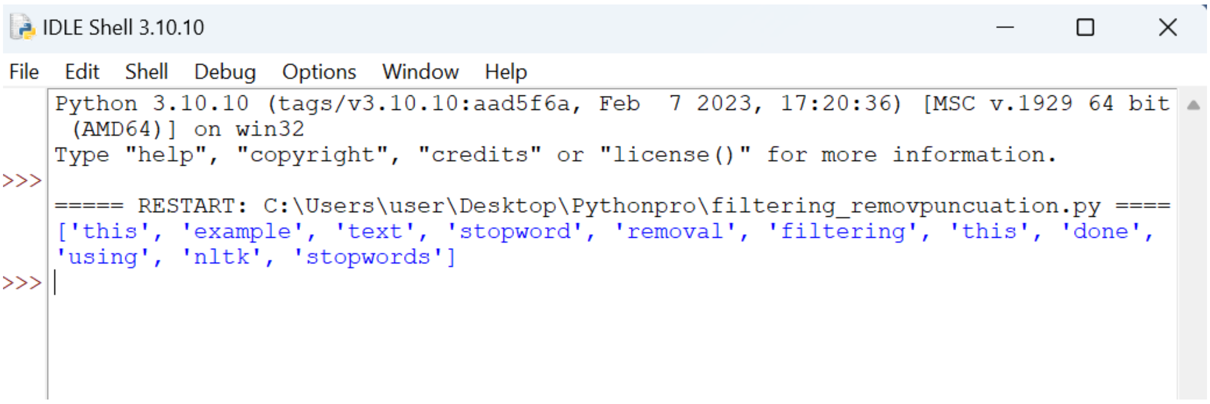
```
File  Edit  Shell  Debug  Options  Window  Help
Python 3.10.10 (tags/v3.10.10:aad5f6a, Feb  7 2023, 17:20:36) [MSC v.1929 64 bit
 (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: C:\Users\user\Desktop\Pythonpro\stopword_remove.py =========
['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', "you're", "
you've", "you'll", "you'd", 'your', 'yours', 'yourself', 'yourselves', 'he', 'hi
m', 'his', 'himself', 'she', "she's", 'her', 'hers', 'herself', 'it', "it's", 'i
ts', 'itself', 'they', 'them', 'their', 'theirs', 'themselves', 'what', 'which',
 'who', 'whom', 'this', 'that', "that'll", 'these', 'those', 'am', 'is', 'are',
'was', 'were', 'be', 'been', 'being', 'have', 'has', 'had', 'having', 'do', 'doe
s', 'did', 'doing', 'a', 'an', 'the', 'and', 'but', 'if', 'or', 'because', 'as',
 'until', 'while', 'of', 'at', 'by', 'for', 'with', 'about', 'against', 'between
', 'into', 'through', 'during', 'before', 'after', 'above', 'below', 'to', 'from
', 'up', 'down', 'in', 'out', 'on', 'off', 'over', 'under', 'again', 'further',
'then', 'once', 'here', 'there', 'when', 'where', 'why', 'how', 'all', 'any', 'b
oth', 'each', 'few', 'more', 'most', 'other', 'some', 'such', 'no', 'nor', 'not'
, 'only', 'own', 'same', 'so', 'than', 'too', 'very', 's', 't', 'can', 'will', '
just', 'don', "don't", 'should', "should've", 'now', 'd', 'll', 'm', 'o', 're',
've', 'y', 'ain', 'aren', "aren't", 'couldn', "couldn't", 'didn', "didn't", 'doe
sn', "doesn't", 'hadn', "hadn't", 'hasn', "hasn't", 'haven', "haven't", 'isn', "
isn't", 'ma', 'mightn', "mightn't", 'mustn', "mustn't", 'needn', "needn't", 'sha
n', "shan't", 'shouldn', "shouldn't", 'wasn', "wasn't", 'weren', "weren't", 'won
', "won't", 'wouldn', "wouldn't"]

Unfiltered:  ['This', 'is', 'an', 'example', 'text', 'for', 'stopword', 'removal
', 'and', 'filtering', '.', 'This', 'is', 'done', 'using', 'NLTK', "'s", 'stopwo
rds', '.']

Filtered:  ['This', 'example', 'text', 'stopword', 'removal', 'filtering', '.',
'This', 'done', 'using', 'NLTK', "'s", 'stopwords', '.']
>>>
```

Removing Punctuations and Word less than 2 Characters

Program

```
import nltk
import string
text = "This is an example text for stopword removal and filtering. This is done using NLTK's stopwords."
words = nltk.word_tokenize(text)
stopwords = nltk.corpus.stopwords.words("english")
# Extending the stopwords list
stopwords.extend(string.punctuation)
# Remove stop words and tokens with length < 2
cleaned = [word.lower() for word in words if (word not in stopwords) and len(word) > 2]
print(cleaned)
```

Output



Filtering Profanity

```
Banned_List = ["idiot", "stupid", "donkey"]
sentence = "You are not only stupid , but also an idiot ."
def censor(sentence = ""):
    new_sentence = ""

    for word in sentence.split():
        print(word)
        if word in Banned_List:
            new_sentence += '* '
```

```
    else:
        new_sentence += word + ' '

    return new_sentence
print(censor(sentence))
```

Output

```
IDLE Shell 3.10.10
File  Edit  Shell  Debug  Options  Window  Help
Python 3.10.10 (tags/v3.10.10:aad5f6a, Feb  7 2023, 17:20:36) [MSC v.1929 64 bit
 (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
======== RESTART: C:\Users\user\Desktop\Pythonpro\filtering_profanity.py =======
You
are
not
only
stupid
,
but
also
an
idiot
.
You are not only * , but also an * .
>>>
```

Conclusion: In the above experiment we have studied regarding preprocessing of text in detail like filtration, stop word removal, tokenization and have tried to implement the code for it and have successfully executed it.

Experiment No. 2

Morphological Analysis

Aim:

To Study Morphological Analysis and apply techniques like Stemming and Lemmatization.

Theory:

Morphological analysis is a field of linguistics that studies the structure of words. It identifies how a word is produced through the use of morphemes. A morpheme is a basic unit of the English language. The morpheme is the smallest element of a word that has grammatical function and meaning. Free morpheme and bound morpheme are the two types of morphemes. A single free morpheme can become a complete word.

For instance, a bus, a bicycle, and so forth. A bound morpheme, on the other hand, cannot stand alone and must be joined to a free morpheme to produce a word. ing, un, and other bound morphemes are examples.

Inflectional Morphology and Derivational Morphology are the two types of morphology. Both of these types have their own significance in various areas related to the Natural Language Processing.

While performing the morphological analysis, each particular word is analyzed. Non-word tokens such as punctuation are removed from the words. Hence the remaining words are assigned categories. For instance, Ram's iPhone cannot convert the video from .mkv to .mp4. In Morphological analysis, word by word the sentence is analyzed. So here, Ram is a proper noun, Ram's is assigned as possessive suffix and .mkv and .mp4 is assigned as a file extension. As shown above, the sentence is analyzed word by word. Each word is assigned a syntactic category. The file extensions are also identified present in the sentence which is behaving as an adjective in the above example. In the above example, the possessive suffix is also identified. This is a very important step as the judgment of prefixes and suffixes will depend on a syntactic category for the word. For example, swims and swims are different. One makes it plural, while the other makes it a third-person singular verb. If the prefix or suffix is incorrectly interpreted then the meaning and understanding of the sentence are completely changed. The interpretation assigns a category to the word. Hence, discard the uncertainty from the word.

Morphological Parsing

It is the process of determining the morphenes from which a given word is constructed. Morphenes are the smallest meaningful words which cannot be divided further. Morphenes can be stem or afix. Stem are the root word whereas afix can be prefix, suffix or infix. For example-

Unsuccessfull → un success ful

(prefix) (stem) (suffix)

Stemming

Stemming is the process of producing morphological variants of a root/base word. Stemming programs are commonly referred to as stemming algorithms or stemmers. Stemming is a natural language processing technique that lowers inflection in words to their root forms, hence aiding in the preprocessing of text, words, and documents for text normalization.

For example, the words “programming,” “programmer,” and “programs” can all be reduced down to the common word stem “program.” In other words, “program” can be used as a synonym for the prior three inflection words.

Stemming algorithms function by taking a list of frequent prefixes and suffixes found in inflected words and chopping off the end or beginning of the word. This can occasionally result in word stems that are not real words; thus, we can affirm this approach certainly has its pros, but it’s not without its limitations.

Types of Stemmer in NLTK

There are several kinds of stemming algorithms, and all of them are included in Python NLTK. They are

Porter Stemmer – PorterStemmer()

Snowball Stemmer – SnowballStemmer()

Lancaster Stemmer – LancasterStemmer()

Regexp Stemmer – RegexpStemmer()

Porter Stemmer – PorterStemmer()

Martin Porter invented the Porter Stemmer or Porter algorithm in 1980. Five steps of word reduction are used in the method, each with its own set of mapping rules. Porter Stemmer is the original stemmer and is renowned for its ease of use and rapidity.

Snowball Stemmer – SnowballStemmer()
Martin Porter also created Snowball Stemmer. The method utilized in this instance is more precise and is referred to as "English Stemmer" or "Porter2 Stemmer." It is somewhat faster and more logical than the original Porter Stemmer.

Lancaster Stemmer – LancasterStemmer()
Lancaster Stemmer is straightforward, although it often produces results with excessive stemming. Over-stemming renders stems non-linguistic or meaningless.

Regexp Stemmer – RegexpStemmer()
Regex stemmer identifies morphological affixes using regular expressions. Substrings matching the regular expressions will be discarded.

Lemmatization

Lemmatization is similar to Stemming, however, a Lemmatizer always returns a valid word. Stemming uses rules to cut the word, whereas a Lemmatizer searched for the root word, also called as Lemma from the WordNet. Moreover, lemmatization takes care of converting a word into its base form; i.e. words like am, is, are will be converted to "be".
Stemming vs Lemmatization

Word	Stemming	Lemmatization
information	inform	information
informative	inform	informative
computers	comput	computer
feet	feet	foot

Lemmatization can be implemented using the following python packages,
Wordnet Lemmatizer, Spacy Lemmatizer, TextBlob, CLiPS Pattern, Stanford CoreNLP, Gensim Lemmatizer, TreeTagger

Porter Stemmer

Program

```
import string
from nltk.tokenize import word_tokenize
from nltk.stem import PorterStemmer
ps = PorterStemmer()
example_sentence = "A stemmer for English operating on the stem cat should identify such strings as cats, catlike, and catty. A stemming algorithm might also reduce the words fishing, fished, and fisher to the stem fish. The stem need not be a word, for example the Porter algorithm reduces, argue, argued, argues, arguing, and argus to the stem argu."
# Remove punctuation
example_sentence_no_punct = example_sentence.translate(str.maketrans("", "", string.punctuation))
# Create tokens
word_tokens = word_tokenize(example_sentence_no_punct)
# Perform stemming
print("{0:20}{1:20}".format("--Word--","--Stem--"))
for word in word_tokens:
   print ("{0:20}{1:20}".format(word, ps.stem(word)))
```

Output

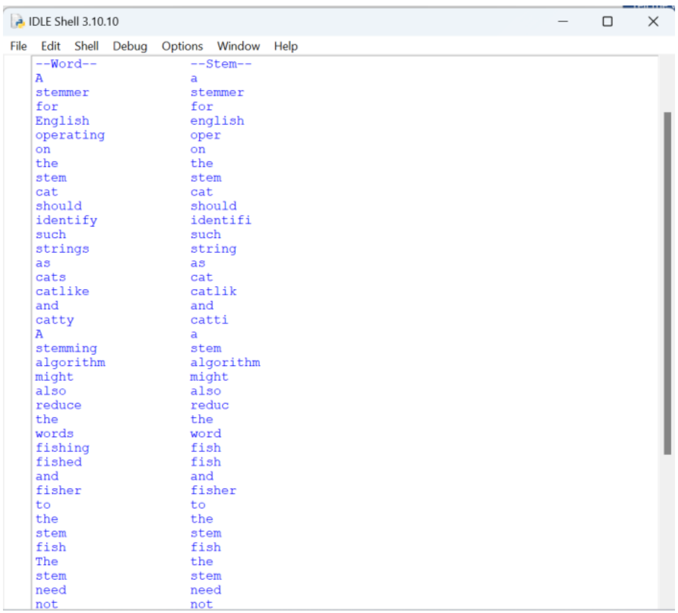
```
IDLE Shell 3.10.10
File  Edit  Shell  Debug  Options  Window  Help
Python 3.10.10 (tags/v3.10.10:aad5f6a, Feb  7 2023, 17:20:36) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
============ RESTART: C:\Users\user\Desktop\Pythonpro\stem_porter.py ===========
--Word--            --Stem--
A                   a
stemmer             stemmer
for                 for
English             english
operating           oper
on                  on
the                 the
stem                stem
cat                 cat
should              should
identify            identifi
such                such
strings             string
as                  as
cats                cat
catlike             catlik
and                 and
catty               catti
A                   a
stemming            stem
algorithm           algorithm
might               might
also                also
reduce              reduc
the                 the
words               word
fishing             fish
fished              fish
and                 and
fisher              fisher
to                  to
the                 the
stem                stem
fish                fish
The                 the
stem                stem
```

Snowball Stemmer

Program

```
import string
from nltk.tokenize import word_tokenize
from nltk.stem import SnowballStemmer
snowball = SnowballStemmer(language='english')
example_sentence = "A stemmer for English operating on the stem cat should identify such strings as cats, catlike, and catty. A stemming algorithm might also reduce the words fishing, fished, and fisher to the stem fish. The stem need not be a word, for example the Porter algorithm reduces, argue, argued, argues, arguing, and argus to the stem argu."
# Remove punctuation
example_sentence_no_punct = example_sentence.translate(str.maketrans("", "", string.punctuation))
# Create tokens
word_tokens = word_tokenize(example_sentence_no_punct)
# Perform stemming
print("{0:20}{1:20}".format("--Word--","--Stem--"))
for word in word_tokens:
    print ("{0:20}{1:20}".format(word, snowball.stem(word)))
```

Output

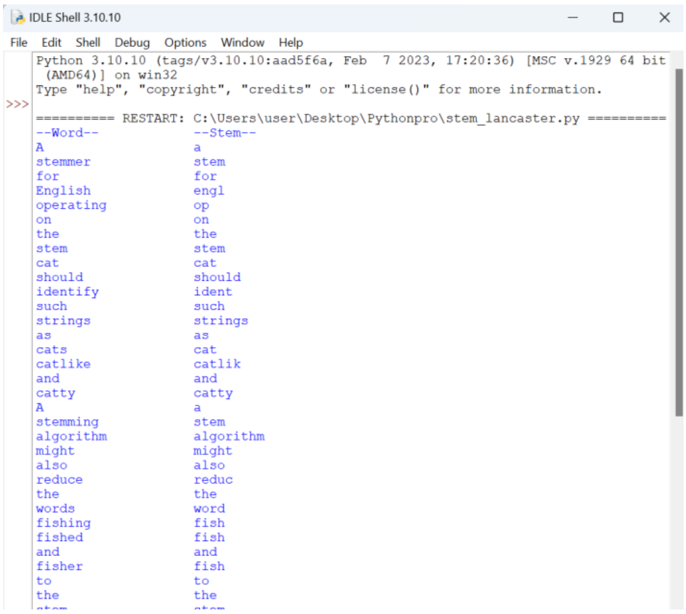


Lancaster Stemmer

Program

```
import string
from nltk.tokenize import word_tokenize
from nltk.stem import LancasterStemmer
ls = LancasterStemmer()
example_sentence = "A stemmer for English operating on the stem cat should identify such strings as cats, catlike, and catty. A stemming algorithm might also reduce the words fishing, fished, and fisher to the stem fish. The stem need not be a word, for example the Porter algorithm reduces, argue, argued, argues, arguing, and argus to the stem argu."
# Remove punctuation
example_sentence_no_punct = example_sentence.translate(str.maketrans("", "", string.punctuation))
# Create tokens
word_tokens = word_tokenize(example_sentence_no_punct)
# Perform stemming
print("{0:20}{1:20}".format("--Word--","--Stem--"))
for word in word_tokens:
    print ("{0:20}{1:20}".format(word, ls.stem(word)))
```

Output



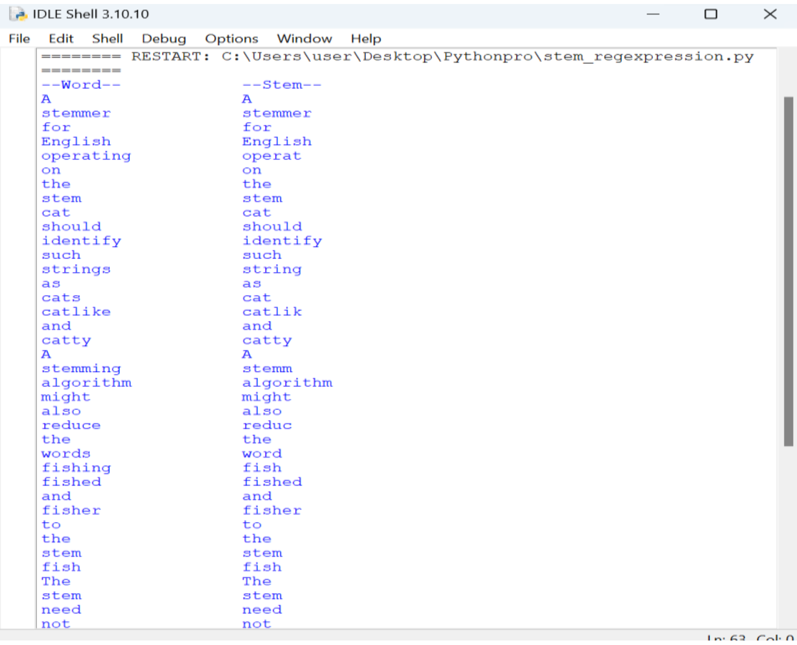
```
IDLE Shell 3.10.10
File  Edit  Shell  Debug  Options  Window  Help
Python 3.10.10 (tags/v3.10.10:aad5f6a, Feb  7 2023, 17:20:36) [MSC v.1929 64 bit
 (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: C:\Users\user\Desktop\Pythonpro\stem_lancaster.py ==========
--Word--            --Stem--
A                   a
stemmer             stem
for                 for
English             engl
operating           op
on                  on
the                 the
stem                stem
cat                 cat
should              should
identify            ident
such                such
strings             strings
as                  as
cats                cat
catlike             catlik
and                 and
catty               catty
A                   a
stemming            stem
algorithm           algorithm
might               might
also                also
reduce              reduc
the                 the
words               word
fishing             fish
fished              fish
and                 and
fisher              fish
to                  to
the                 the
```

Regexp Stemmer

Program

```
import string
from nltk.tokenize import word_tokenize
from nltk.stem import RegexpStemmer
regexp = RegexpStemmer('ing$|s$|e$|able$', min=4)
example_sentence = "A stemmer for English operating on the stem cat should identify such strings as cats, catlike, and catty. A stemming algorithm might also reduce the words fishing, fished, and fisher to the stem fish. The stem need not be a word, for example the Porter algorithm reduces, argue, argued, argues, arguing, and argus to the stem argu."
# Remove punctuation
example_sentence_no_punct = example_sentence.translate(str.maketrans("", "", string.punctuation))
# Create tokens
word_tokens = word_tokenize(example_sentence_no_punct)
# Perform stemming
print("{0:20}{1:20}".format("--Word--","--Stem--"))
for word in word_tokens:
    print ("{0:20}{1:20}".format(word, regexp.stem(word)))
```

Output



Comparison of various stemmer

```
from nltk.stem import PorterStemmer, SnowballStemmer, LancasterStemmer, RegexpStemmer
porter = PorterStemmer()
lancaster = LancasterStemmer()
snowball = SnowballStemmer(language='english')
regexp = RegexpStemmer('ing$|s$|e$|able$', min=4)
word_list = ["friend", "friendship", "friends", "friendships"]
print("{0:20}{1:20}{2:20}{3:30}{4:40}".format("Word","Porter Stemmer","Snowball Stemmer","Lancaster Stemmer",'Regexp Stemmer'))
for word in word_list:
print("{0:20}{1:20}{2:20}{3:30}{4:40}".format(word,porter.stem(word),snowball.stem(word),lancaster.stem(word),regexp.stem(word)))
```

Output

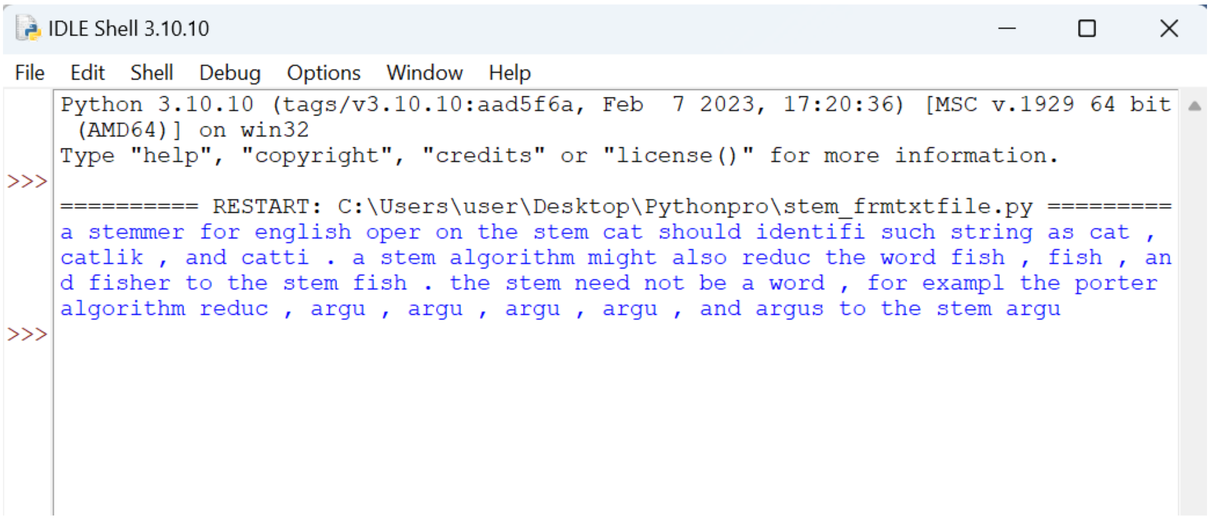
```
IDLE Shell 3.10.10
File  Edit  Shell  Debug  Options  Window  Help
Python 3.10.10 (tags/v3.10.10:aad5f6a, Feb  7 2023, 17:20:36) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
=========== RESTART: C:\Users\user\Desktop\Pythonpro\stem_cmparisn.py ==========
Word                Porter Stemmer      Snowball Stemmer    Lancaster Stemmer             Regexp Stemmer
friend              friend              friend              friend                        friend
friendship          friendship          friendship          friend                        friendship
friends             friend              friend              friend                        friend
friendships         friendship          friendship          friend                        friendship
>>>
```

Stemming from Textfile

```
from nltk.tokenize import word_tokenize
from nltk.stem import SnowballStemmer
def stemming(text):
    snowball = SnowballStemmer(language='english')
    list=[]
    for token in word_tokenize(text):
        list.append(snowball.stem(token))
    return ' '.join(list)
```

```
with open('text_file.txt') as f:
    text=f.read()
print(stemming(text))
```

Output

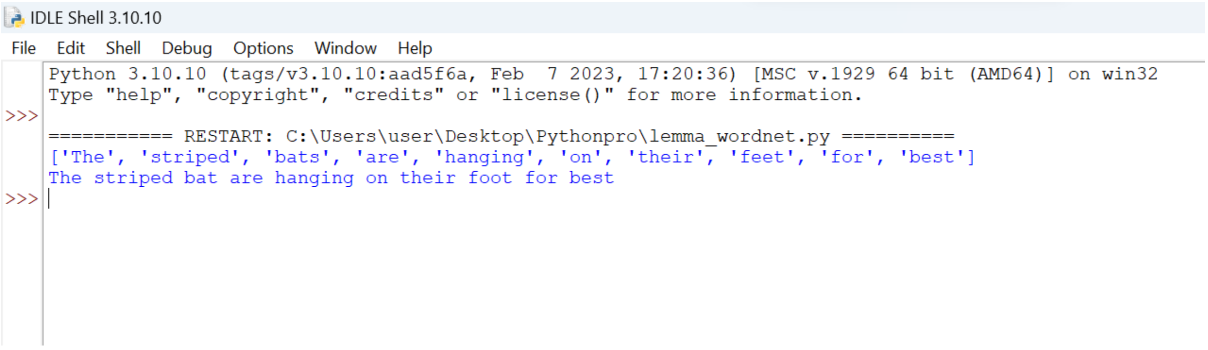


Lemmatization

Wordnet Lemmatizer with NLTK

```
import nltk
from nltk.stem import WordNetLemmatizer
# Init the Wordnet Lemmatizer
lemmatizer = WordNetLemmatizer()
sentence = "The striped bats are hanging on their feet for best"
# Tokenize: Split the sentence into words
word_list = nltk.word_tokenize(sentence)
print(word_list)
#> ['The', 'striped', 'bats', 'are', 'hanging', 'on', 'their', 'feet', 'for', 'best']
# Lemmatize list of words and join
lemmatized_output = ' '.join([lemmatizer.lemmatize(w) for w in word_list])
print(lemmatized_output)
```

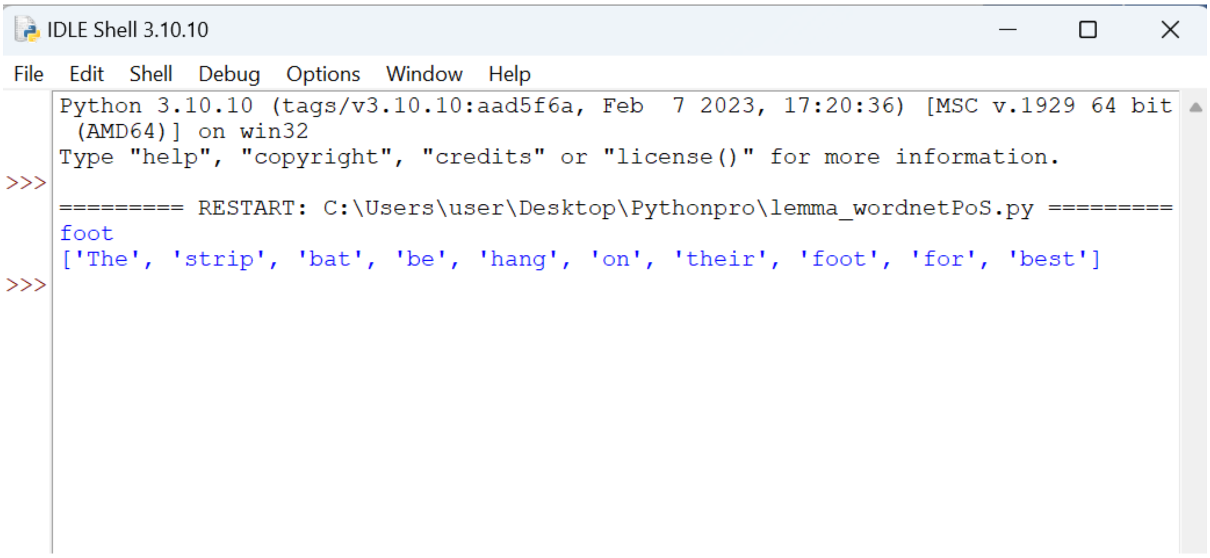
Output



Wordnet Lemmatizer using POS Tag

```
import nltk
from nltk.stem import WordNetLemmatizer
from nltk.corpus import wordnet
def get_wordnet_pos(word):
    """Map POS tag to first character lemmatize() accepts"""
    tag = nltk.pos_tag([word])[0][1][0].upper()
    tag_dict = {"J": wordnet.ADJ,
                "N": wordnet.NOUN,
                "V": wordnet.VERB,
                "R": wordnet.ADV}
    return tag_dict.get(tag, wordnet.NOUN)
# 1. Init Lemmatizer
lemmatizer = WordNetLemmatizer()
# 2. Lemmatize Single Word with the appropriate POS tag
word = 'feet'
print(lemmatizer.lemmatize(word, get_wordnet_pos(word)))
# 3. Lemmatize a Sentence with the appropriate POS tag
sentence = "The striped bats are hanging on their feet for best"
print([lemmatizer.lemmatize(w,get_wordnet_pos(w)) for w in nltk.word_tokenize(sentence)])
```

Output



Conclusion: Thus, in the above experiment we have studied regarding morphological analysis in detail with stemming, lemmatization, regular expression and successfully implemented using various python packages and tested.

Experiment No. 3

N-Gram Model

Aim:

To study N-gram model and implement python program to simulate the functions of N-Gram model using NLTK and other packages.

Theory:

N-grams are contiguous sequences of n items (typically words) from a given text. N-grams can be used in natural language processing and machine learning to extract features from text data. The most common types of n-grams are bigrams (n=2), trigrams (n=3), and 4-grams (n=4).

Given a sequence of N-1 words, an N-gram model predicts the most probable word that might follow this sequence. It's a probabilistic model that's trained on a corpus of text. Such a model is useful in many NLP applications including speech recognition, machine translation and predictive text input.

An N-gram model is built by counting how often word sequences occur in corpus text and then estimating the probabilities. Since a simple N-gram model has limitations, improvements are often made via smoothing, interpolation and backoff.

An N-gram model is one type of a Language Model (LM), which is about finding the probability distribution over word sequences.

Consider two sentences: "There was heavy rain" vs. "There was heavy flood". From experience, we know that the former sentence sounds better. An N-gram model will tell us that "heavy rain" occurs much more often than "heavy flood" in the training corpus. Thus, the first sentence is more probable and will be selected by the model.

A model that simply relies on how often a word occurs without looking at previous words is calledunigram. If a model considers only the previous word to predict the current word, then it's calledbigram. If two previous words are considered, then it's a trigram model.

An n-gram model for the above example would calculate the following probability:

P('There was heavy rain') = P('There', 'was', 'heavy', 'rain') = P('There')P('was'|'There')P('heavy'|'There was')P('rain'|'There was heavy')

Since it's impractical to calculate these conditional probabilities, using Markov assumption, we approximate this to a bigram model:
P('There was heavy rain') ~ P('There')P('was'|'There')P('heavy'|'was')P('rain'|'heavy')

In speech recognition, input may be noisy and this can lead to wrong speech-to-text conversions. N-gram models can correct this based on their knowledge of the probabilities. Likewise, N-gram models are used in machine translation to produce more natural sentences in the target language.

When correcting for spelling errors, sometimes dictionary lookups will not help. For example, in the phrase "in about fifteen mineuts" the word 'minuets' is a valid dictionary word but it's incorrect in this context. N-gram models can correct such errors.
N-gram models are usually at word level. It's also been used at character level to do stemming, that is, separate the rootword from the suffix. By looking at N-gram statistics, we could also classify languages or differentiate between US and UK spellings. For example, 'sz' is common in Czech; 'gb' and 'kp' are common in Igbo.

In general, many NLP applications benefit from N-gram models including part-of-speech tagging, natural language generation, word similarity, sentiment extraction and predictive text input

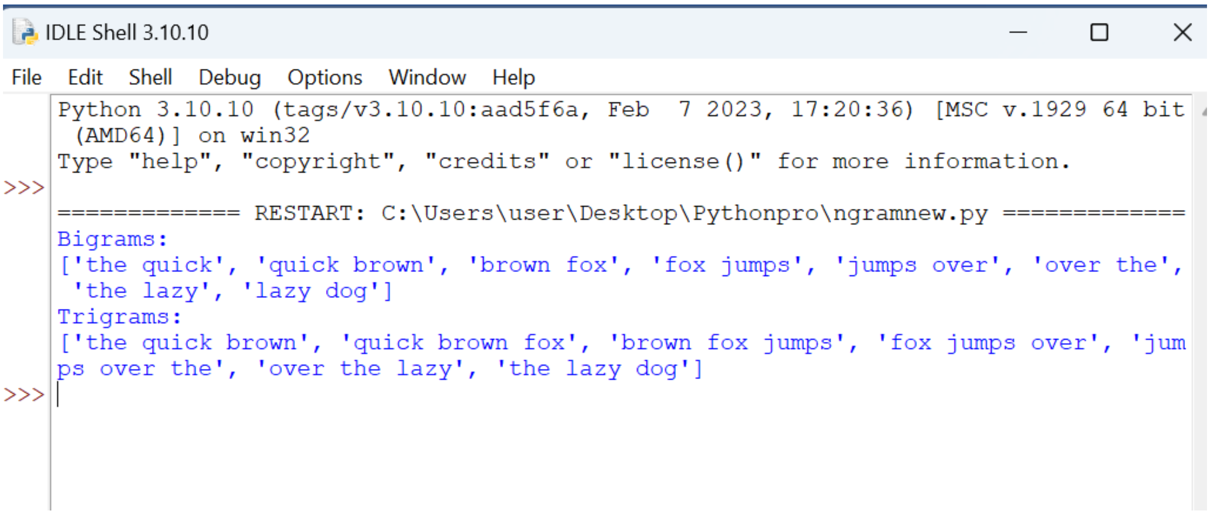
Program

N-Grams using Regex

```
import re
text = "The quick brown fox jumps over the lazy dog"
# Function to generate n-grams
def generate_ngrams(text, n):
   # Convert text to lowercase and remove punctuation
   text = text.lower()
   text = re.sub(r'[^\w\s]','',text)
   # Split text into words
```

```
    words = text.split()
    # Generate n-grams
    ngrams = []
    for i in range(len(words)-n+1):
        ngram = ' '.join(words[i:i+n])
        ngrams.append(ngram)
    return ngrams
# Generate and print bigrams
print("Bigrams:")
print(generate_ngrams(text, 2))
# Generate and print trigrams
print("Trigrams:")
print(generate_ngrams(text, 3))
```

Output



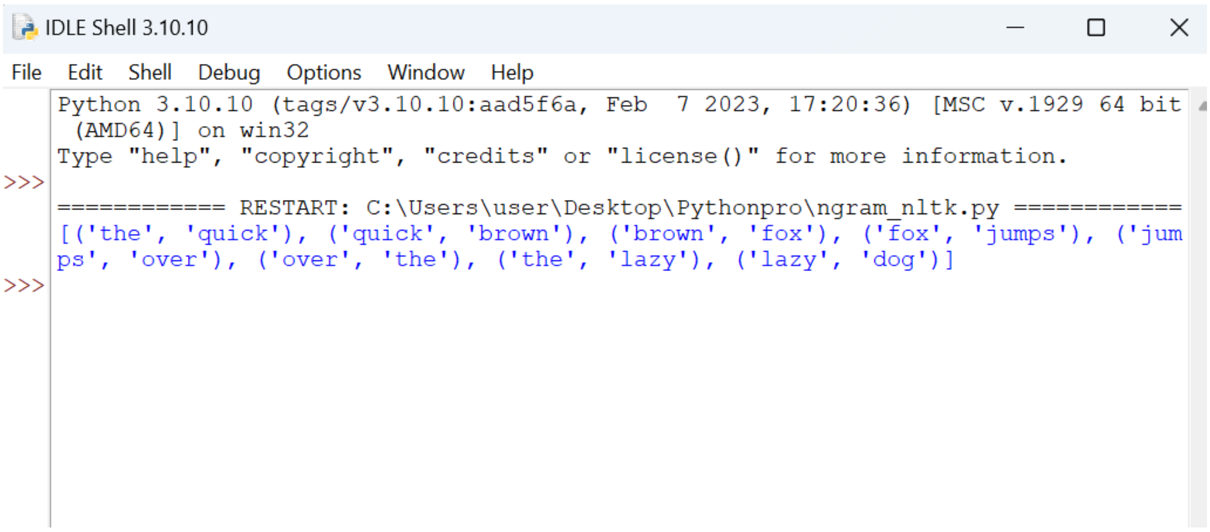
N-Grams using NLTK

Program

```
import nltk
from nltk.util import ngrams
text = "The quick brown fox jumps over the lazy dog"
n = 2 # Generate bigrams
tokens = nltk.word_tokenize(text.lower())
bigrams = ngrams(tokens, n)
```

print(list(bigrams))

Output



Conclusion:

Thus, in the above experiment we have studied regarding N-Gram Model in detail with the help of theory and then tried to implement the code and successfully executed it.

Experiment No.4

POS Tagging

Aim:

To study POS tagging and implement it in python for a given sentence.

Theory:

Part-of-speech (POS) tagging is a process in natural language processing (NLP) where each word in a text is labeled with its corresponding part of speech. This can include nouns, verbs, adjectives, and other grammatical categories.

POS tagging is useful for a variety of NLP tasks, such as information extraction, named entity recognition, and machine translation. It can also be used to identify the grammatical structure of a sentence and to disambiguate words that have multiple meanings.

Default tagging is a basic step for the part-of-speech tagging. It is performed using the Default Tagger class. The Default Tagger class takes 'tag' as a single argument. NN is the tag for a singular noun. Default Tagger is most useful when it gets to work with most common part-of-speech tag. that's why a noun tag is recommended.

Tagging is a kind of classification that may be defined as the automatic assignment of description to the tokens. Here the descriptor is called tag, which may represent one of the part-of-speech, semantic information and so on.

Now, if we talk about Part-of-Speech (PoS) tagging, then it may be defined as the process of assigning one of the parts of speech to the given word. It is generally called POS tagging. In simple words, we can say that POS tagging is a task of labelling each word in a sentence with its appropriate part of speech. We already know that parts of speech include nouns, verb, adverbs, adjectives, pronouns, conjunction and their sub-categories.

Most of the POS tagging falls under Rule Base POS tagging, Stochastic POS tagging and Transformation based tagging

Rule-based POS Tagging

One of the oldest techniques of tagging is rule-based POS tagging. Rule-based taggers use dictionary or lexicon for getting possible tags for tagging each word. If the word has more than one possible tag, then rule-based taggers use hand-written rules to identify the correct tag. Disambiguation can also be performed in rule-based tagging by analyzing the linguistic features of a word along with its preceding as well as following words. For example, suppose if the preceding word of a word is article, then word must be a noun.

Stochastic POS Tagging

Another technique of tagging is Stochastic POS Tagging. Now, the question that arises here is which model can be stochastic. The model that includes frequency or probability (statistics) can be called stochastic. Any number of different approaches to the problem of part-of-speech tagging can be referred to as stochastic tagger.

The simplest stochastic tagger applies the following approaches for POS tagging –

Word Frequency Approach

In this approach, the stochastic taggers disambiguate the words based on the probability that a word occurs with a particular tag. We can also say that the tag encountered most frequently with the word in the training set is the one assigned to an ambiguous instance of that word. The main issue with this approach is that it may yield inadmissible sequence of tags.

Tag Sequence Probabilities

It is another approach of stochastic tagging, where the tagger calculates the probability of a given sequence of tags occurring. It is also called n-gram approach. It is called so because the best tag for a given word is determined by the probability at which it occurs with the n previous tags.

Transformation-based Tagging

Transformation based tagging is also called Brill tagging. It is an instance of the transformation-based learning (TBL), which is a rule-based algorithm for automatic tagging of POS to the given text. TBL, allows us to have linguistic knowledge in a readable form, transforms one state to another state by using transformation rules.

It draws the inspiration from both the previous explained taggers − rule-based and stochastic. If we see similarity between rule-based and transformation tagger, then like rule-based, it is also based on the rules that specify what tags need to be assigned to what words. On the other hand, if we see similarity between stochastic and transformation tagger then like stochastic, it is machine learning technique in which rules are automatically induced from data.

HMM for POS Tagging

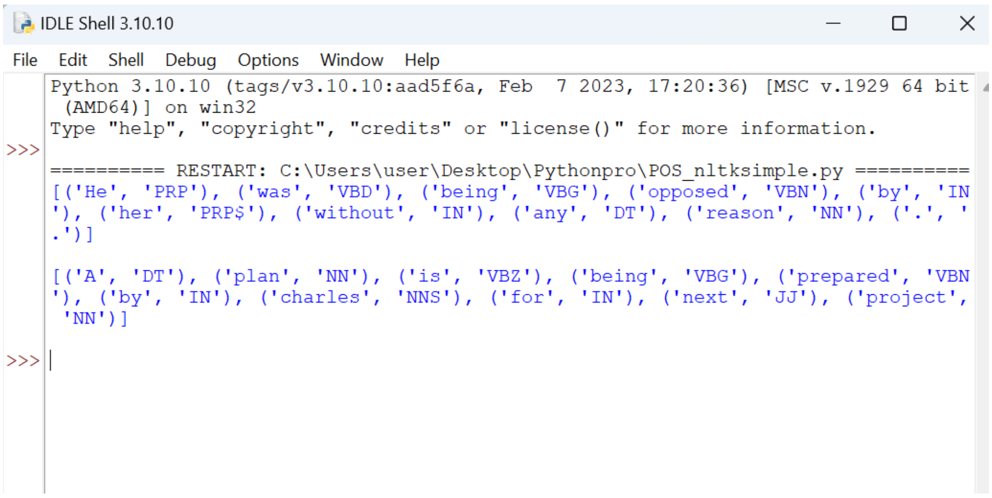
The POS tagging process is the process of finding the sequence of tags which is most likely to have generated a given word sequence. We can model this POS process by using a Hidden Markov Model (HMM), where tags are the hidden states that produced the observable output, i.e., the words.

NLTK Pos Tagger

Program

```
import nltk
from nltk.tokenize import word_tokenize, sent_tokenize
sentence = "He was being opposed by her without any reason.\
 A plan is being prepared by charles for next project"
for sent in sent_tokenize(sentence):
    wordtokens = word_tokenize(sent)
    print(nltk.pos_tag(wordtokens),end='\n\n')
```

Output



NLTK POS using Spacy

```
import spacy
nlp = spacy.load('en_core_web_sm')
sentence = "He was being opposed by her without any reason.\
```

A plan is being prepared by charles for next project"

for token in nlp(sentence):

print(f'{token.text:{10}} {token.tag_:>{10}}\t{spacy.explain(token.tag_):<{50}} {token.pos_:>{5}} {spacy.explain(token.pos_):<{50}}')

Output

```
IDLE Shell 3.10.10
File  Edit  Shell  Debug  Options  Window  Help
Python 3.10.10 (tags/v3.10.10:aad5f6a, Feb  7 2023, 17:20:36) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
============= RESTART: C:\Users\user\Desktop\Pythonpro\pos_spacy.py ============
He                 PRP   pronoun, personal                                  PRON pronoun
was                VBD   verb, past tense                                    AUX auxiliary
being              VBG   verb, gerund or present participle                  AUX auxiliary
opposed            VBN   verb, past participle                              VERB verb
by                  IN   conjunction, subordinating or preposition           ADP adposition
her                PRP   pronoun, personal                                  PRON pronoun
without             IN   conjunction, subordinating or preposition           ADP adposition
any                 DT   determiner                                          DET determiner
reason              NN   noun, singular or mass                             NOUN noun
.                    .   punctuation mark, sentence closer                 PUNCT punctuation
A                   DT   determiner                                          DET determiner
plan                NN   noun, singular or mass                             NOUN noun
is                 VBZ   verb, 3rd person singular present                   AUX auxiliary
being              VBG   verb, gerund or present participle                  AUX auxiliary
prepared           VBN   verb, past participle                              VERB verb
by                  IN   conjunction, subordinating or preposition           ADP adposition
charles            NNS   noun, plural                                       NOUN noun
for                 IN   conjunction, subordinating or preposition           ADP adposition
next                JJ   adjective (English), other noun-modifier (Chinese)  ADJ adjective
project             NN   noun, singular or mass                             NOUN noun
>>>
```

Programs using Visual

NLTK POS Tagging using Spacy Visual

import spacy

from spacy import displacy

nlp = spacy.load('en_core_web_sm')

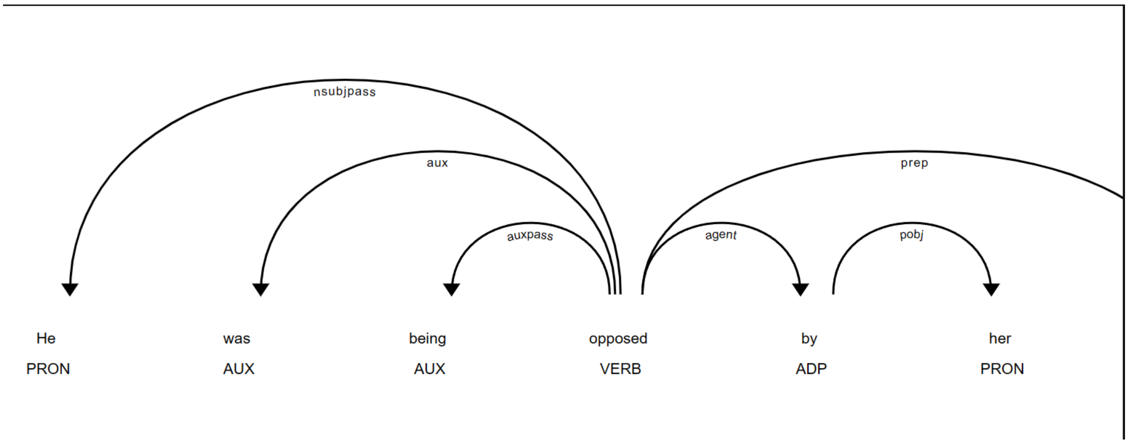
doc =nlp("He was being opposed by her without any reason.")

displacy.render(doc, style='dep')

for token in nlp(doc):

print(f'{token.text:{10}} {token.tag_:>{10}}\t{spacy.explain(token.tag_):<{50}} {token.pos_:>{5}} {spacy.explain(token.pos_):<{50}}')

Output



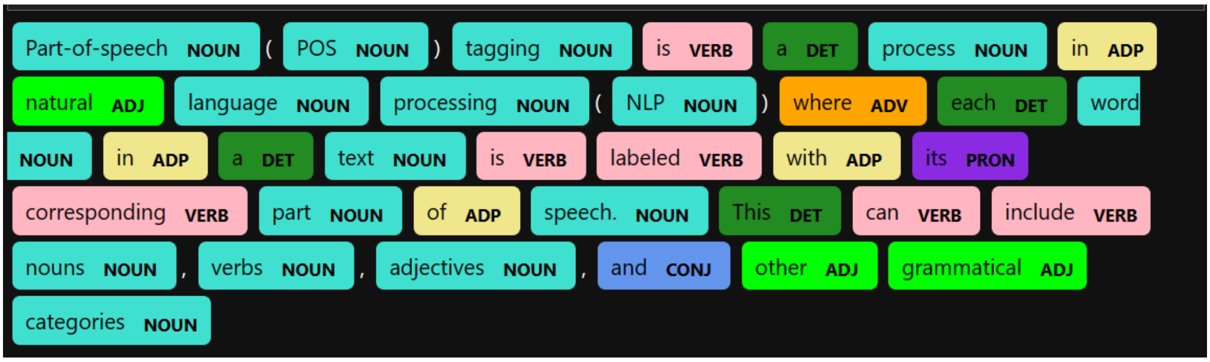
He	PRP	pronoun, personal	PRON pronoun
was	VBD	verb, past tense	AUX auxiliary
being	VBG	verb, gerund or present participle	AUX auxiliary
opposed	VBN	verb, past participle	VERB verb
by	IN	conjunction, subordinating or preposition	ADP adposition
her	PRP	pronoun, personal	PRON pronoun
without	IN	conjunction, subordinating or preposition	ADP adposition
any	DT	determiner	DET determiner
reason	NN	noun, singular or mass	NOUN noun
.	.	punctuation mark, sentence closer	PUNCT punctuation

Program

```
import spacy
from spacy import displacy
import nltk
from nltk.tokenize import TreebankWordTokenizer as twt
#text = "Call me Ishmael."
def visualize_pos(text):
    pos_tags = ["PRON", "VERB", "NOUN", "ADJ", "ADP",
              "ADV", "CONJ", "DET", "NUM", "PRT"]
      # Tokenize text and pos tag each token
    tokens = twt().tokenize(text)
    tags = nltk.pos_tag(tokens, tagset = "universal")

    # Get start and end index (span) for each token
```

```
span_generator = twt().span_tokenize(text)
spans = [span for span in span_generator]

# Create dictionary with start index, end index,
# pos_tag for each token
ents = []
for tag, span in zip(tags, spans):
    if tag[1] in pos_tags:
        ents.append({"start" : span[0],
                     "end" : span[1],
                     "label" : tag[1] })

doc = {"text" : text, "ents" : ents}

colors = {"PRON": "blueviolet",
          "VERB": "lightpink",
          "NOUN": "turquoise",
          "ADJ" : "lime",
          "ADP" : "khaki",
          "ADV" : "orange",
          "CONJ" : "cornflowerblue",
          "DET" : "forestgreen",
          "NUM" : "salmon",
          "PRT" : "yellow"}

options = {"ents" : pos_tags, "colors" : colors}

displacy.render(doc,
                style = "ent",
                options = options,
                manual = True,
               )
```

visualize_pos("Part-of-speech (POS) tagging is a process in natural language processing (NLP) where each word in a text is labeled with its corresponding part of speech. This can include nouns, verbs, adjectives, and other grammatical categories.")



Conclusion: Thus, we have studied POS Tagging in the above experiment also learned regarding different types of POS Tagging and tried to implement the code for POS Tagging and successfully executed it.

Experiment No. 5

Chunking

Aim:

To study Chunking and implement chunking using nltk pos tags and regular expression and to visualize the chunked words.

Theory:

Chunk extraction or partial parsing is a process of meaningful extracting short phrases from the sentence (tagged with Part-of-Speech).Chunks are made up of words and the kinds of words are defined using the part-of-speech tags. One can even define a pattern or words that can't be a part of chuck and such words are known as chinks. A Chunk Rule class specifies what words or patterns to include and exclude in a chunk.

Defining Chunk patterns:

Chuck patterns are normal regular expressions which are modified and designed to match the part-of-speech tag designed to match sequences of part-of-speech tags. Angle brackets are used to specify an indiviual tag for example –to match a noun tag. One can define multiple tags in the same way.

Chunking is a process of extracting phrases from unstructured text. Instead of just simple tokens which may not represent the actual meaning of the text, its advisable to use phrases such as "South Africa" as a single word instead of 'South' and 'Africa' separate words.

Chunking in NLP is Changing a perception by moving a "chunk", or a group of bits of information, in the direction of a Deductive or Inductive conclusion through the use of language. Chunking up or down allows the speaker to use certain language patterns, to utilize the natural internal process through language, to reach for higher meanings or search for more specific bits/portions of missing information.

When we "Chunk Up" the language gets more abstract and there are more chances for agreement, and when we "Chunk Down" we tend to be looking for the specific details that may have been missing in the chunk up.

As an example, if you ask the question “for what purpose cars?” you may get the answer “transport”, which is a higher chunk and more toward abstract.

If you asked “what specifically about a car”? you will start to get smaller pieces of information about a car.

Lateral thinking will be the process of chunking up and then looking for other examples: For example, “for what intentions cars?”, “transportation”, “what are other examples of transportation?” “Buses!”

The chunk to be extracted is defined using regex (regular expressions) along with the POS_tags. From regex, we’ll mainly use the following:

? = 0 or 1 match of the preceding expression

* = 0 or more match of the preceding expression

+ = 1 or more match of the preceding expression

. = specifies any single character except a new line character

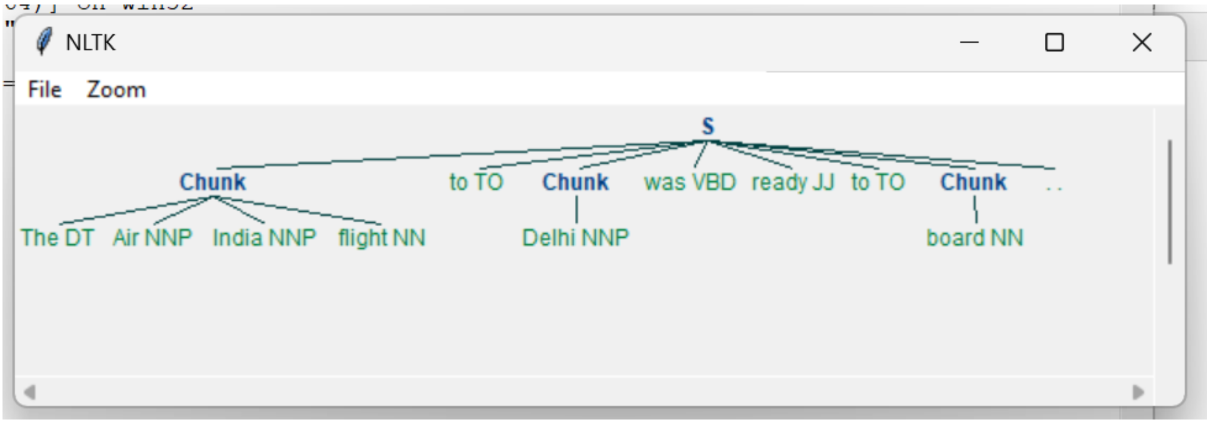
For e.g. to extract all the proper nouns present in a sentence, one of the chunks that can be used is r’’’ Chunk: {<DT>*<NNP>*<NN>*} ’’’ (where ‘<>’ denotes a POS_tag).

Program

```
import nltk
from nltk import pos_tag
from nltk import word_tokenize
from nltk import RegexpParser
# Example sentence
text = " The Air India flight to Delhi was ready to board."
# Splitiing the sentence into words
list_of_words = word_tokenize(text)
# Applying POS_tagging
tagged_words = pos_tag(list_of_words)
chunk_to_be_extracted = r''' Chunk: {<DT>*<NNP>*<NN>*} '''
#chunk_to_be_extracted = r''' Chunk: {<VBD>*} '''
# Applying chunking to the text
```

```
chunkParser = nltk.chunk.RegexpParser(chunk_to_be_extracted)
chunked_sentence = chunkParser.parse(tagged_words)
chunked_sentence.draw()
print('Chunks obtained: \n')
for subtree in chunked_sentence.subtrees():
    if subtree.label() == 'Chunk':
        print(subtree)
```

Output



Conclusion:

Thus, in the above experiment we have studies regarding chunking and tried to implement the code for same and successfully executed it.

Experiment No. 6

Named Entity Recognition

Aim:

To study Named Entity Recognition and write a python program to categorize and display various NER tags.

Theory:

Named Entity Recognition (NER) is a standard NLP problem which involves spotting named entities (people, places, organizations etc.) from a chunk of text, and classifying them into a predefined set of categories. Some of the practical applications of

NER include:

- Scanning news articles for the people, organizations and locations reported
- Providing concise features for search optimization: instead of searching the entire content, one may simply search for the major entities involved.
- Quickly retrieving geographical locations talked about in Twitter posts.

How NER works

At the heart of any NER model is a two step process:

Detect a named entity

Categorize the entity

Beneath this lie a couple of things.

Step one involves detecting a word or string of words that form an entity. Each word represents a token: “The Great Lakes” is a string of three tokens that represents one entity. Inside-outside-beginning taggingis a common way of indicating where entities begin and end. We’ll explore this further in a future blog post.

The second step requires the creation of entity categories.

How is NER used?

NER is suited to any situation in which a high-level overview of a large quantity of text is helpful. With NER, you can, at a glance, understand the subject or theme of a body of text and quickly group texts based on their relevancy or similarity. Some notable NER use cases include:

Human resources

Speed up the hiring process by summarizing applicants' CVs; improve internal workflows by categorizing employee complaints and questions.

Customer support

Improve response times by categorizing user requests, complaints and questions and filtering by priority keywords.

Different Blocks Present in A Typical NER Model

- A typically named entity recognition NLP model consists of several components, including:
- Tokenization: Tokenization breaks text into individual tokens (usually words or punctuation marks).
- Part-of-speech tagging: Labelling each token with its corresponding part of speech (e.g. noun, verb, adjective, etc.).
- Chunking: Group tokens into "chunks" based on their part-of-speech tags.
- Name entity recognition: Identifying named entities and classifying them into predefined categories.
- Entity disambiguation: The process of determining the correct meaning of a named entity, especially when multiple entities with the same name are present in the text.

How does Named Entity Recognition Work?

Several approaches can be used to perform named entity recognition NLP models. The most common methods include the following:

- Rule-based methods use a set of predefined rules and patterns to identify named entities in text.
- Statistical methods use a probabilistic framework to identify named entities in a text by training a model on a large annotated text corpus.

- Machine learning methods also use probabilistic frameworks but rely on Machine Learning algorithms to learn the patterns in the data.

Once the model is trained, we can identify named entities in a new text by applying the learned patterns and features.

Some popular tools and libraries for implementing NER include:

- Stanford NER: A Java-based NER toolkit developed by Stanford University.
- spaCy: A Python library for NLP tasks, including NER.
- NLTK: A Python library for NLP tasks, including NER.
- OpenNLP: An Apache-licensed NLP library written in Java.

NER Using NLTK

Program

```
import nltk
from nltk.tokenize import word_tokenize
from nltk.tag import pos_tag
from nltk.chunk import ne_chunk
sentence = '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'
ne_tree = ne_chunk(pos_tag(word_tokenize(sentence)))
print(ne_tree)
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'
def preprocess(sent):
    sent = nltk.word_tokenize(sent)
    sent = nltk.pos_tag(sent)
    return sent
sent = preprocess(ex)
sent
print(sent)

pattern = 'NP: {<DT>?<JJ>*<NN>}'
cp = nltk.RegexpParser(pattern)
```

cs = cp.parse(sent)

print(cs)

NPChunker = nltk.RegexpParser(pattern)

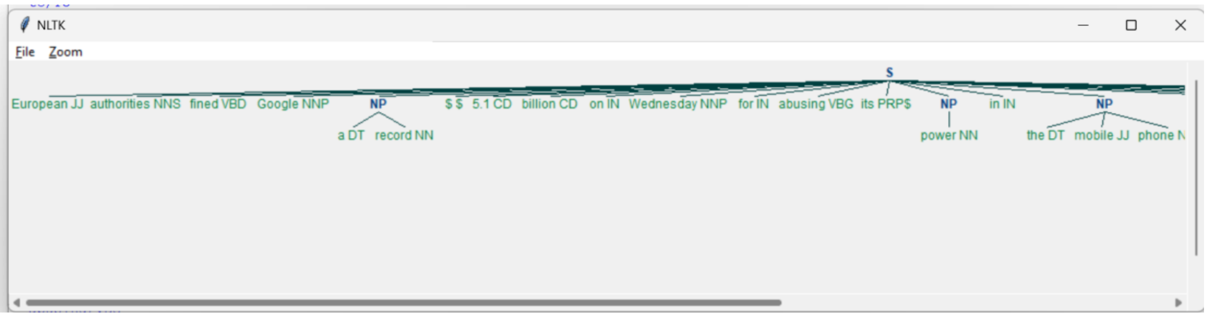
result = NPChunker.parse(sent)

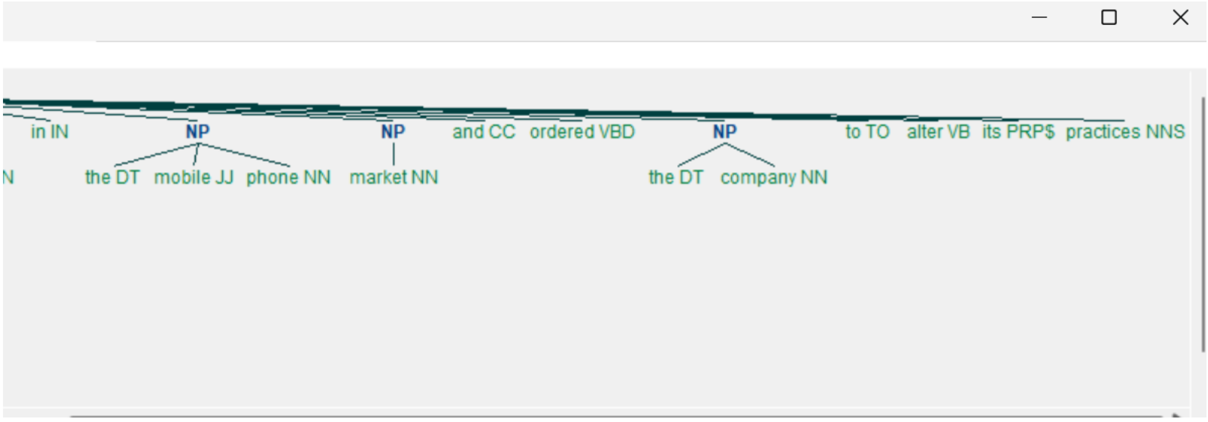
result.draw()

Output

```
*IDLE Shell 3.10.10*
File  Edit  Shell  Debug  Options  Window  Help
Python 3.10.10 (tags/v3.10.10:aad5f6a, Feb  7 2023, 17:20:36) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
============== RESTART: C:\Users\user\Desktop\Pythonpro\ner1.py ==============
(S
  (GPE European/JJ)
  authorities/NNS
  fined/VBD
  (PERSON Google/NNP)
  a/DT
  record/NN
  $/$
  5.1/CD
  billion/CD
  on/IN
  Wednesday/NNP
  for/IN
  abusing/VBG
  its/PRP$
  power/NN
  in/IN
  the/DT
  mobile/JJ
  phone/NN
  market/NN
  and/CC
  ordered/VBD
  the/DT
  company/NN
  to/TO
  alter/VB
  its/PRP$
  practices/NNS)
[('European', 'JJ'), ('authorities', 'NNS'), ('fined', 'VBD'), ('Google', 'NNP'), ('a', 'DT'), ('record', 'NN'), ('$', '$'), ('5.1', 'CD'), ('billion', '
CD'), ('on', 'IN'), ('Wednesday', 'NNP'), ('for', 'IN'), ('abusing', 'VBG'), ('its', 'PRP$'), ('power', 'NN'), ('in', 'IN'), ('the', 'DT'), ('mobile', 'J
J'), ('phone', 'NN'), ('market', 'NN'), ('and', 'CC'), ('ordered', 'VBD'), ('the', 'DT'), ('company', 'NN'), ('to', 'TO'), ('alter', 'VB'), ('its', 'PRP$
'), ('practices', 'NNS')]
(S
```

```
  European/JJ
  authorities/NNS
  fined/VBD
  Google/NNP
  (NP a/DT record/NN)
  $/$
  5.1/CD
  billion/CD
  on/IN
  Wednesday/NNP
  for/IN
  abusing/VBG
  its/PRP$
  (NP power/NN)
  in/IN
  (NP the/DT mobile/JJ phone/NN)
  (NP market/NN)
  and/CC
  ordered/VBD
  (NP the/DT company/NN)
  to/TO
  alter/VB
  its/PRP$
  practices/NNS)
```





Program using Spacy

```
from pprint import pprint
import spacy
from spacy import displacy
from collections import Counter
import en_core_web_sm
nlp = en_core_web_sm.load()
doc = nlp("On June 12th, 2022, Lionel Messi, a famous Argentine soccer player, led the
Argentine National Soccer Team to a stunning victory against Brazil in the final match of the
FIFA World Cup held at the Maracana Stadium in Rio de Janeiro, Brazil. Messi scored two
goals, earning him the title of the tournament's top scorer. Fans from all over the world,
including a large group of Argentine supporters, cheered on the team in their native languages
throughout the game. Adidas, the official sponsor of the World Cup, provided the official match
ball, which was used throughout the tournament. The game lasted for 120 minutes, with
Argentina ultimately winning 3-2 in a thrilling overtime shootout. The victory marked the first
time Argentina had won the World Cup since 1986 and resulted in a prize money of $38 million
for the team. The entire country celebrated the victory, and the team's achievement was later
commemorated through a mural depicting Messi and his teammates in Buenos Aires,
```

Argentina. When asked about his performance he replied in Spanish that the movie, GOAL was his inspiration. He loves yoga of Hindu tradition and follows Libertarian Party in politics. Apart from sports he loves to raft in the river Indus and likes to hike in Everest and Mount Kilimanjaro. He reads guerrilla warfare by Ernesto Che Guevara and he spent 30% of his college days to ride Toyoto Camry for about 20 miles per day.")

```
pprint([(X.text, X.label_) for X in doc.ents])
#pprint([(X, X.ent_iob_, X.ent_type_) for X in doc])
print(len(doc.ents))
labels = [x.label_ for x in doc.ents]
print(Counter(labels))
items = [x.text for x in doc.ents]
print(Counter(items).most_common(3))
sentences = [x for x in doc.sents]
print(sentences)
displacy.render(nlp(str(sentences)), jupyter=True, style='ent')
```

Output

```
[('June 12th, 2022', 'DATE'),
 ('Messi', 'PERSON'),
 ('Argentine', 'NORP'),
 ('the Argentine National Soccer Team', 'ORG'),
 ('Brazil', 'GPE'),
 ('the FIFA World Cup', 'EVENT'),
 ('the Maracana Stadium', 'FAC'),
 ('Rio de Janeiro', 'GPE'),
 ('Brazil', 'GPE'),
 ('Messi', 'PERSON'),
 ('two', 'CARDINAL'),
 ('Argentine', 'NORP'),
 ('Adidas', 'PERSON'),
 ('the World Cup', 'EVENT'),
 ('120 minutes', 'TIME'),
 ('Argentina', 'GPE'),
 ('3-2', 'CARDINAL'),
 ('first', 'ORDINAL'),
 ('Argentina', 'GPE'),
 ('the World Cup', 'EVENT'),
 ('1986', 'DATE'),
 ('$38 million', 'MONEY'),
 ('Messi', 'PERSON'),
 ('Buenos Aires', 'GPE'),
 ('Argentina', 'GPE'),
 ('Spanish', 'LANGUAGE'),
 ('Hindu', 'NORP')
```

```
('Hindu', 'NORP'),
('Libertarian Party', 'ORG'),
('Indus', 'GPE'),
('Everest', 'LOC'),
('Mount Kilimanjaro', 'LOC'),
('Ernesto Che Guevara', 'PERSON'),
('30%', 'PERCENT'),
('Toyoto Camry', 'ORG'),
('about 20 miles', 'QUANTITY')]
35
Counter({'GPE': 8, 'PERSON': 5, 'NORP': 3, 'ORG': 3, 'EVENT': 3, 'DATE': 2, 'CARDINAL': 2, 'LOC': 2, 'FAC': 1, 'TIME': 1, 'ORDINAL': 1, 'MONE
Y': 1, 'LANGUAGE': 1, 'PERCENT': 1, 'QUANTITY': 1})
[('Messi', 3), ('Argentina', 3), ('Argentine', 2)]
```

```
[On June 12th, 2022, Lionel Messi, a famous Argentine soccer player, led the Argentine National Soccer Team to a stunning victory against Brazi
l in the final match of the FIFA World Cup held at the Maracana Stadium in Rio de Janeiro, Brazil., Messi scored two goals, earning him the tit
le of the tournament's top scorer., Fans from all over the world, including a large group of Argentine supporters, cheered on the team in their
native languages throughout the game., Adidas, the official sponsor of the World Cup, provided the official match ball, which was used througho
ut the tournament., The game lasted for 120 minutes, with Argentina ultimately winning 3-2 in a thrilling overtime shootout., The victory marke
d the first time Argentina had won the World Cup since 1986 and resulted in a prize money of $38 million for the team., The entire country cele
brated the victory, and the team's achievement was later commemorated through a mural depicting Messi and his teammates in Buenos Aires, Argent
ina., When asked about his performance he replied in Spanish that the movie, GOAL was his inspiration., He loves yoga of Hindu tradition and fo
llows Libertarian Party in politics., Apart from sports he loves to raft in the river Indus and likes to hike in Everest and Mount Kilimanjar
o., He reads guerrilla warfare by Ernesto Che Guevara and he spent 30% of his college days to ride Toyoto Camry for about 20 miles per day.]
```

[On June 12th, 2022 DATE , Lionel Messi PERSON , a famous Argentine NORP soccer player, led the Argentine National Soccer Team ORG to a stunning victory against Brazil GPE in the final match of the FIFA World Cup EVENT held at the Maracana Stadium FAC in Rio de Janeiro GPE , Brazil GPE ., Messi PERSON scored two CARDINAL goals, earning him the title of the tournament's top scorer., Fans from all over the world, including a large group of Argentine NORP supporters, cheered on the team in their native languages throughout the game., Adidas PERSON , the official sponsor of the World Cup EVENT , provided the official match ball, which was used throughout the tournament., The game lasted for 120 minutes TIME , with Argentina GPE ultimately winning 3-2 CARDINAL in a thrilling overtime shootout., The victory marked the first ORDINAL time Argentina GPE had won the World Cup EVENT since 1986 DATE and resulted in a prize money of $38 million MONEY for the team., The entire country celebrated the victory, and the team's achievement was later commemorated through a mural depicting Messi PERSON and his teammates in Buenos Aires GPE , Argentina GPE ., When asked about his performance he replied in Spanish LANGUAGE that the movie, GOAL was his inspiration., He loves yoga of Hindu NORP tradition and follows Libertarian Party ORG in politics., Apart from sports he loves to raft in the river Indus GPE and likes to hike in Everest LOC and Mount Kilimanjaro LOC ., He reads guerrilla warfare by Ernesto Che Guevara PERSON and he spent 30% PERCENT of his college days to ride Toyoto Camry ORG for about 20 miles QUANTITY per day.]

Conclusion:

Thus, in the above experiment we have studied regarding named entity recognition, working of named entity recognition, how named entity recognition can be used and then implemented the code for the same and successfully executed it.

Experiment No. 7

Word Generation

Aim:

To implement python program to automatically generate words using various corpora.

In Natural Language Processing (NLP), a word generator is a type of algorithm or model that generates words or text based on certain criteria.

A common use case of word generators in NLP is to generate text that is similar to existing text or to a particular style or genre. For example, a language model trained on Shakespearean text might be used to generate new text that resembles the style and vocabulary of Shakespeare.

Word generators can also be used in language translation and language learning applications. In these contexts, a word generator might be used to generate example sentences or to suggest appropriate vocabulary based on the user's input.

There are various techniques used in NLP for word generation, including Markov chains, recurrent neural networks, and transformer models. The quality of the generated text depends on the complexity of the algorithm and the quality and quantity of the training data used to train the model.

Another type of word generator is a generative adversarial network (GAN), which uses a combination of a generator network and a discriminator network to learn to generate realistic text. The generator network generates fake text samples, while the discriminator network tries to distinguish between the fake and real text. The two networks are trained in an adversarial manner, with the generator trying to fool the discriminator and the discriminator trying to correctly identify the fake text.

Word generators can be used for a variety of NLP tasks, including text generation, language translation, and sentiment analysis. The quality of the generated text depends on the complexity and accuracy of the underlying model, as well as the quality and quantity of the training data used to train the model. In general, more sophisticated models and larger datasets tend to produce better results.

Corpus:

A corpus is a collection of texts that is used for linguistic analysis and research. In NLTK, the corpus module provides access to a wide range of corpora in various languages, including:

- The Brown Corpus: a collection of 500 texts in American English, totaling about 1 million words, that are representative of different genres and styles of writing.
- The Gutenberg Corpus: a collection of over 25,000 free electronic books, mostly in English, that are in the public domain.
- The Reuters Corpus: a collection of over 10,000 news articles from the Reuters news agency, classified into 90 topics.
- The WordNet Corpus: a lexical database of English words and their semantic relationships, organized into synsets (sets of synonyms) and hierarchies of hypernyms (broader concepts) and hyponyms (narrower concepts).

Program

Word-Generator using Corpus in Wordnet:

```
import nltk

from nltk.corpus import wordnet

import random

# Get a list of synsets from the WordNet corpus

synsets = list(wordnet.all_synsets())

# Generate a random word

def generate_word():

    # Select a random synset

    synset = random.choice(synsets)

    # Select a random lemma from the synset

    lemma = random.choice(synset.lemmas())
```

```
        # Get the name of the lemma
        word = lemma.name().replace('_', ' ')
        return word
    # Generate multiple random words
    for i in range(50):
        print(generate_word())
```

Output

```
========================= RESTART: D:\Python\WordGen.py ========================
connotative
penicillin V potassium
cassette deck
wobble
Senecio cineraria
hit
interchange
var
rigorously
gratuitously
uranium 238
alpha globulin
Pacific spiny dogfish
generality
cyclopean masonry
Hamamelis vernalis
gunman
Anomala orientalis
stockholder of record
narrate
local
wood stork
asparagus
economical
mamey
temporal artery
salad burnet
Herr
showy milkweed
polyamide
inscrutability
torturously
train of thought
Mohammed Ali
macule
order Tinamiformes
manoeuvre
Fenusa
```

Word Generator using corpus in brown:

```
import nltk

from nltk.corpus import brown

nltk.download('brown')

brown_words = brown.words()

# Train a language model on the Brown corpus

def train_language_model(words):

    text = nltk.Text(words)

    bigrams = list(nltk.bigrams(text))

    language_model = nltk.ConditionalFreqDist(bigrams)

    return language_model

# Generate a list of words using a language model

def generate_words_from_model(model, start_word, num_words):

    words = [start_word]

    for i in range(num_words-1):

        next_word = model[words[-1]].max()

        words.append(next_word)

    return words

# Example usage:

model = train_language_model(brown_words)

generated_words = generate_words_from_model(model, 'is', 50)

print(generated_words)
```

Output

```
======================== RESTART: D:\Python\WordGen2.py ========================
[nltk_data] Downloading package brown to
[nltk_data]     C:\Users\sivab\AppData\Roaming\nltk_data...
[nltk_data]   Package brown is already up-to-date!
['is', 'a', 'few', 'years', 'ago', ',', 'and', 'the', 'same', 'time', ',', 'and'
, 'the', 'same', 'time', ',', 'and', 'the', 'same', 'time', ',', 'and', 'the', '
same', 'time', ',', 'and', 'the', 'same', 'time', ',', 'and', 'the', 'same', 'ti
me', ',', 'and', 'the', 'same', 'time', ',', 'and', 'the', 'same', 'time', ',',
'and', 'the', 'same', 'time']
>>>
```

Word Generator using corpus in Words:

```
import nltk

from nltk.corpus import words

nltk.download('words')

word_list = words.words()

# Generate a list of n random words

def generate_words(n):

    return [word_list[i] for i in range(n)]

# Example usage:

words = generate_words(50)

print(words)
```

Output

```
======================== RESTART: D:\Python\WordGen3.py ========================
[nltk_data] Downloading package words to
[nltk_data]     C:\Users\sivab\AppData\Roaming\nltk_data...
[nltk_data]   Package words is already up-to-date!
['A', 'a', 'aa', 'aal', 'aalii', 'aam', 'Aani', 'aardvark', 'aardwolf', 'Aaron',
 'Aaronic', 'Aaronical', 'Aaronite', 'Aaronitic', 'Aaru', 'Ab', 'aba', 'Ababdeh'
, 'Ababua', 'abac', 'abaca', 'abacate', 'abacay', 'abacinate', 'abacination', 'a
baciscus', 'abacist', 'aback', 'abactinal', 'abactinally', 'abaction', 'abactor'
, 'abaculus', 'abacus', 'Abadite', 'abaff', 'abaft', 'abaisance', 'abaiser', 'ab
aissed', 'abalienate', 'abalienation', 'abalone', 'Abama', 'abampere', 'abandon'
, 'abandonable', 'abandoned', 'abandonedly', 'abandonee']
>>>
```

Conclusion:

Thus, in the above experiment we have implemented word generation using various corpus like wordnet, brown corpus.

Experiment 8

Sentiment Analysis

Aim

To implement python program to demonstrate the give phrase is positive, negative using sentiment analysis

SENTIMENT ANALYSIS:

Sentiment analysis is used to identify the view or emotion behind a situation. It basically means to analyse and find the emotion or intent behind a piece of text or speech or any mode of communication.

For example, social media platform services like Facebook used to just have two emotions associated with each post, i.e. You can like a post or you can leave the post without any reaction and that basically signifies that you didn't like it.

DATA PRE-PROCESSING:

Now, we will perform some pre-processing on the data before converting it into vectors and passing it to the machine learning model.

We will create a function for pre-processing of data.

1. First, we will iterate through each record, and using a regular expression, we will get rid of any characters apart from alphabets.

2. Then, we will convert the string to lowercase as, the word "Good" is different from the word "good".

Because, without converting to lowercase, it will cause an issue when we will create vectors of these words, as two different vectors will be created for the same word which we don't want to.

3. Then we will check for stop words in the data and get rid of them. Stop words are commonly used words in a sentence such as "the", "an", "to" etc. which do not add much value.

4. Then, we will perform lemmatization on each word, i.e., change the different forms of a word into a single item called a lemma.

A Lemma is a base form of a word. For example, “run”, “running” and “runs” are all forms of the same lexeme, where the “run” is the lemma. Hence, we are converting all occurrences of the same lexeme to their respective lemma.

5. And, then return a corpus of processed data.

But first, we will create an object of WordNetLemmatizer and then we will perform the transformation. Now, we will create a Word Cloud. It is a data visualization technique used to depict text in such a way that, the more frequent words appear enlarged as compared to less frequent words. This gives us a little insight into, how the data looks after being processed through all the steps until now.

BAG OF WORDS

Now, we will use the Bag of Words Model (BOW), which is used to represent the text in the form of a bag of words, i.e., the grammar and the order of words in a sentence are not given any importance, instead, multiplicities. (The number of times a word occurs in a document) is the main point of concern.

Basically, it describes the total occurrence of words within a document.

Scikit-Learn provides a neat way of performing the bag of words technique using Count Vectorizer.

WORKING MODEL:

•Rule-based Sentiment Analysis.

•Automated or Machine Learning Sentiment Analysis.

TEXTBLOB:

Text blob is a Python NLP library that uses a natural language toolkit (NLTK). It uses NLTK because it is simple, easy to deploy, will use up fewer resources, gives dependency parsing, and can be used even for small applications. Text blob can be used for complex analysis and working with textual data.

Features

•Noun phrase extraction

•Part-of-speech tagging

•Sentiment analysis

•Classification (Naive Bayes, Decision Tree)

•Tokenization (splitting text into words and sentences)

•Word and phrase frequencies

•Parsing

•n-grams

•Word inflection (pluralization and singularization) and lemmatization

•Spelling correction

•Add new models or languages through extensions

•WordNet integration

BENEFITS:

Text Blob is a python library and offers a simple API to access its methods and perform basic NLP tasks. A good thing about Text Blob is that they are just like python strings. So, you can transform and play with it same like we did in python.

Program

CODE:

```
from textblob import TextBlob

# Define a function for sentiment analysis

def analyze_sentiment(text):

    """

    Analyzes the sentiment of a given text using TextBlob.
```

```
    Args:
        text (str): The input text to analyze.
    Returns:
        str: The sentiment of the text ('positive', 'negative', or 'neutral').
    """
    # Create a TextBlob object
    blob = TextBlob(text)
    # Get the sentiment polarity (-1 to 1, where -1 is very negative and 1 is very positive)
    polarity = blob.sentiment.polarity
    # Define the threshold for positive and negative sentiment
    positive_threshold = 0.1
    negative_threshold = -0.1
    # Determine the sentiment based on polarity
    if polarity > positive_threshold:
        return 'positive'
    elif polarity < negative_threshold:
        return 'negative'
    else:
        return 'neutral'
# Example usage
text1 = input("Enter a text 1: ")
text2 = input("Enter a text 2: ")
text3 = input("Enter a text 3: ")
```

```
print(analyze_sentiment(text1))  # Output: 'positive'

print(analyze_sentiment(text2))  # Output: 'negative'

print(analyze_sentiment(text3))  # Output: 'positive'
```

Output

```
Enter a text 1: I love this product, it's amazing!
Enter a text 2: I hate the terrible customer service.
Enter a text 3: The weather is nice today.
positive
negative
positive
```

Conclusion

Thus the program for sentiment analysis is successfully tested for the given phrase.

Experiment 9

SPAM Classification

Aim :

To implement python program for spam classification

Introduction:

NLP - Natural language processing (NLP) refers to the branch of computer science—and more specifically, the branch of artificial intelligence or AI—concerned with giving computers the ability to understand text and spoken words in much the same way human beings can.

NLP combines computational linguistics—rule-based modeling of human language—with statistical, machine learning, and deep learning models. Together, these technologies enable computers to process human language in the form of text or voice data and to 'understand' its full meaning, complete with the speaker or writer's intent and sentiment. NLP drives computer programs that translate text from one language to another, respond to spoken commands, and summarize large volumes of text rapidly—even in real time. There's a good chance you've interacted with NLP in the form of voice-operated GPS systems, digital assistants, speech-to-text dictation software, customer service chatbots, and other consumer conveniences. But NLP also plays a growing role in enterprise solutions that help streamline business operations, increase employee productivity, and simplify mission-critical business processes.

NLTK - The Natural Language Toolkit (NLTK) is a platform used for building Python programs that work with human language data for applying in statistical natural language processing (NLP). It contains text processing libraries for tokenization, parsing, classification, stemming, tagging and semantic reasoning. It also includes graphical demonstrations and sample data sets as well as accompanied by a cook book and a book which explains the principles behind the underlying language processing tasks that NLTK supports.

How it works:

This code is a Python script that detects spam emails using machine learning. It uses a dataset of emails labeled as spam or not spam to train a machine learning model that can predict whether an email is spam or not based on its content. The script first reads in a CSV file containing the email data and preprocesses it by removing stop words and vectorizing the text data using CountVectorizer. This means that it converts the text data into a numerical format that can be used by the machine learning algorithm.

It then trains a Multinomial Naive Bayes classifier on the vectorized data. This classifier is a type of machine learning algorithm that is commonly used for text classification tasks like spam detection.

Finally, it tests the classifier on a small set of messages from the NPS Chat Corpus and plots the results using matplotlib. The plot shows the percentage of messages that were correctly classified as spam or not spam .

How does CountVectorizer works:

CountVectorizer is a tool that converts text data into a numerical format that can be used by machine learning algorithms. It does this by first breaking the text into individual words (a process called tokenization) and then counting the number of times each word appears in the text. The resulting counts are then transformed into a numerical vector that can be used as input to a machine learning algorithm.

For example, if we have the following two sentences:

"The quick brown fox jumps over the lazy dog"

"The lazy dog slept in the sun"

CountVectorizer would first tokenize these sentences into individual words and then

count the number of times each word appears in the entire text:

"the": 3

"quick": 1

"brown": 1

"fox": 1

"jumps": 1

"over": 1

"lazy": 2

"dog": 2

"slept": 1

"in": 1

"sun": 1

These counts would then be transformed into a numerical vector that could be used as input to a machine learning algorithm.

How Multinominal Naive Bayes Classifier works:

- The Multinomial Naive Bayes classifier is a type of machine learning algorithm that is commonly used for text classification tasks like spam detection .
- It is suitable for classification with discrete features (e.g., word counts for text classification) .
- The multinomial distribution normally requires integer feature counts, but in practice, fractional counts such as tf-idf may also work .
- The alpha parameter of the classifier controls the smoothing of the probabilities .
- Smoothing is used to avoid zero probabilities when a feature does not occur in the training set . The value of alpha determines the strength of the smoothing .

Program
Code:

```
import nltk
import pandas as pd

from sklearn.feature_extraction.text import CountVectorizer

from sklearn.naive_bayes import MultinomialNB

import matplotlib.pyplot as plt

nltk.download('stopwords')
```

```
nltk.download('punkt')
nltk.download('nps_chat')
stop_words = set(nltk.corpus.stopwords.words('english'))
df = pd.read_csv('spam.csv', encoding='latin-1')
df = df[['v1', 'v2']]
df = df.rename(columns={'v1': 'label', 'v2': 'text'})
df['text'] = df['text'].apply(lambda x: ' '.join([word for word in x.split() if word.lower() not in
stop_words]))
vectorizer = CountVectorizer()
counts = vectorizer.fit_transform(df['text'].values)
classifier = MultinomialNB()
targets = df['label'].values
classifier.fit(counts, targets)
messages = nltk.corpus.nps_chat.xml_posts()[:1000]
spam_count = 0
not_spam_count = 0
for message in messages:
message_counts = vectorizer.transform([message.text])
if classifier.predict(message_counts) == 'spam':
spam_count += 1
else:
not_spam_count += 1
labels = ['Spam', 'Not Spam']
```

sizes = [spam_count, not_spam_count]

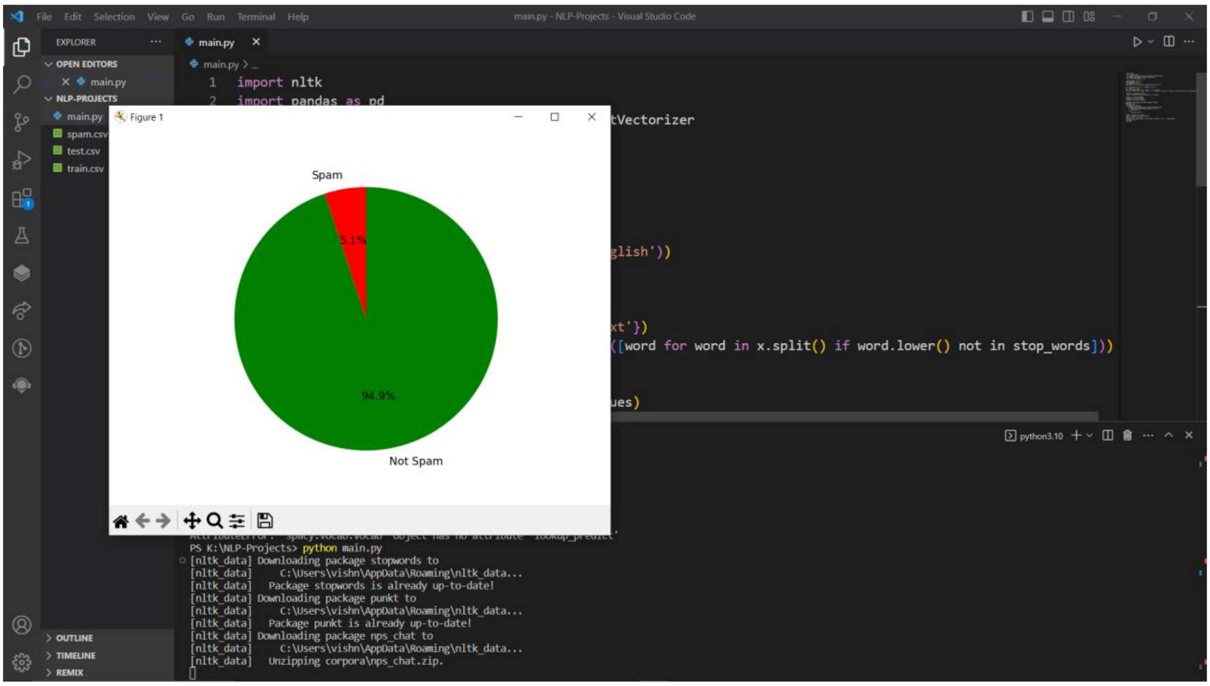
colors = ['red', 'green']

plt.pie(sizes, labels=labels, colors=colors, autopct='%1.1f%%', startangle=90)

plt.axis('equal')

plt.show()

Output



Conclusion

Thus the program for spam classification has successfully implemented.

Experiment 10

Autocorrect

Aim

To develop python program to implement autocorrect functionality using various python libraries

An autocorrect system is a software tool that automatically corrects spelling mistakes and typos in the text. It is commonly used in word processing programs, email clients, and mobile devices.

Code Overview

The code implements an autocorrect system that can suggest corrected words for misspelled words based on their probability of occurrence in a given text corpus. The code has several steps:

Step 1: Data Preprocessing

In this step, the code reads a corpus of text, converts all the text to lowercase, and extracts a list of words from the text using regular expressions. The code also creates a vocabulary set of all the unique words in the text.

Step 2: Calculating Word Frequencies

The get_count function is used to calculate the frequency of each word in the text. It returns a dictionary where each key is a word and each value is the number of times that word appears in the text.

Step 3: Calculating Word Probabilities

The get_probs function is used to calculate the probability of each word in the text. It returns a dictionary where each key is a word and each value is the probability of that word occurring in the text. The probability is calculated as the frequency of the word divided by the total number of words in the text.

Step 4: Generating Possible Corrections The autocorrect system uses four types of edit operations to generate possible corrections for a given misspelled word: delete a letter, switch two adjacent letters, replace a letter with another letter, and insert a letter. These edit operations are implemented in the DeleteLetter, SwitchLetter, replace_letter, and insert_letter functions, respectively. The edit_one_letter function generates a set of possible corrections for a given misspelled word by applying one of the four edit operations. The edit_two_letters function generates a set of possible corrections by applying two edit operations. Both functions return a set of strings representing the corrected words.

Step 5: Selecting Best Corrections

The get_corrections function is used to select the best correction(s) for a given misspelled word. It takes the misspelled word, the word probabilities, the vocabulary set, and the number of suggestions to return as input parameters.

The function first checks if the misspelled word is already in the vocabulary. If it is, the function returns the misspelled word as the only suggestion.

If the misspelled word is not in the vocabulary, the function generates a set of possible corrections by applying the edit operations. It then selects the suggestions that are in the vocabulary and returns them in descending order of their probability.

Step 6: Testing the Autocorrect System

The code prompts the user to enter a word and displays up to three possible corrections. It uses the get_corrections function to generate the suggestions and the word probabilities calculated earlier to rank the suggestions.

Implementation

```
# Step 1: Data Preprocessing
import re # regular expression
from collections import Counter
import numpy as np
import pandas as pd
w = [] #words
with open('sample.txt','r',encoding="utf8") as f:
file_name_data = f.read()
file_name_data = file_name_data.lower()
w = re.findall('\w+', file_name_data)
v = set(w) #vocabulary
#Step 2: Calculating Word Frequencies
def get_count(words):
word_count = {}
```

```
for word in words:
if word in word_count:
word_count[word] += 1
else:
word_count[word] = 1
return word_count
#Step 3: Calculating Word Probabilities
word_count = get_count(w)
def get_probs(word_count_dict):
probs = {}
m = sum(word_count_dict.values())
for key in word_count_dict.keys():
probs[key] = word_count_dict[key] / m
return probs
#Step 4: Generating Possible Corrections
# Delete Letter:
def DeleteLetter(word):
delete_list = []
split_list = []
for i in range(len(word)):
split_list.append((word[0:i], word[i:]))
for a, b in split_list:
delete_list.append(a + b[1:])
return delete_list
delete_word_l = DeleteLetter(word="cans")
# Switch Letter:
def SwitchLetter(word):
split_l = []
switch_l = []
for i in range(len(word)):
split_l.append((word[0:i], word[i:]))
switch_l = [a + b[1] + b[0] + b[2:] for a, b in split_l if len(b) >= 2]
return switch_l
switch_word_l = SwitchLetter(word="eta")
```

```
# Replace letter:
def replace_letter(word):
split_l = []
replace_list = []
for i in range(len(word)):
split_l.append((word[0:i], word[i:]))
alphabets = 'abcdefghijklmnopqrstuvwxyz'
replace_list = [a + l + (b[1:] if len(b) > 1 else '') for a, b in split_l if b for l in alphabets]
return replace_list
replace_l = replace_letter(word='can')
# Insert letter:
def insert_letter(word):
split_l = []
insert_list = []
for i in range(len(word) + 1):
split_l.append((word[0:i], word[i:]))
letters = 'abcdefghijklmnopqrstuvwxyz'
insert_list = [a + l + b for a, b in split_l for l in letters]
# print(split_l)
return insert_list
# Edit one and two letters
def edit_one_letter(word, allow_switches=True):
edit_set1 = set()
edit_set1.update(DeleteLetter(word))
if allow_switches:
edit_set1.update(SwitchLetter(word))
edit_set1.update(replace_letter(word))
edit_set1.update(insert_letter(word))
return edit_set1
def edit_two_letters(word, allow_switches=True):
edit_set2 = set()
edit_one = edit_one_letter(word, allow_switches=allow_switches)
for w in edit_one:
if w:
```

```
edit_two = edit_one_letter(w, allow_switches=allow_switches)
edit_set2.update(edit_two)
return edit_set2
#Step 5: Selecting Best Corrections
def get_corrections(word, probs, vocab, n=2):
suggested_word = []
best_suggestion = []
suggested_word = list(
(word in vocab and word) or edit_one_letter(word).intersection(vocab) or
edit_two_letters(word).intersection(
vocab))
best_suggestion = [[s, probs[s]] for s in list(reversed(suggested_word))]
return best_suggestion
#Step 6: Testing the Autocorrect System
my_word = input("Enter any word:")
probs = get_probs(word_count)
tmp_corrections = get_corrections(my_word, probs, v, 2)
for i, word_prob in enumerate(tmp_corrections):
if(i<3):
print(f"{i+1}: {word_prob[0]}")
else:
break
```

Output:

Enter any word:mezn

1: mean

Enter any word:cwn

1: can

2: own

Enter any word:evrm

1: ever

2: even

3: firm

Aim

Conclusion

Thus the python program to get a word from user with mis-spellings and the words can be corrected to the right words has been successfully developed.

Mini-Project

Chat bot

Aim

To write python program to demonstrate the operation of a chat bot.

A chatbot is an artificial intelligence-powered piece of software in a device (Siri, Alexa, Google Assistant, etc.), application, website, or other networks. It gauges consumer's needs and then assists them in performing a particular task like a commercial transaction, hotel booking, form submission, etc. Today almost every company has a chatbot deployed to engage with the users. Some of the ways in which companies are using chatbots are:

● To deliver flight information

● to connect customers and their finances

● As customer support

The possibilities are (almost) limitless.

Building the Bot

Pre-requisites

Hands-On knowledge of scikit library and NLTK is assumed. However, if you are new to NLP, you can still read the article and then refer back to resources.

Downloading and installing NLTK

1. Install NLTK: run pip install nltk

2. Test installation: run python then type import nltk

For platform-specific instructions, read here.

Installing NLTK Packages

import NLTK and run nltk.download().This will open the NLTK downloader from where you can choose the corpora and models to download. You can also download all packages at once.

Text Pre- Processing with NLTK

The main issue with text data is that it is all in text format (strings). However, Machine learning algorithms need some sort of numerical feature vector to perform the task. So

before we start with any NLP project, we need to pre-process it to make it ideal for work.

Basic text pre-processing includes:

Converting the entire text into uppercase or lowercase so that the algorithm does not

treat the same words in different cases as different

Tokenization: Tokenization is just the term used to describe the process of converting the normal text strings into a list of tokens, i.e., words that we want. A sentence tokenizer can be used to find the list of sentences, and a Word tokenizer can be used to find the list of words in strings.

The NLTK data package includes a pre-trained Punkt tokenizer for English.

Removing Noise, i.e., everything that isn't in a standard number or letter.

Removing Stop words. Sometimes, some extremely common words that appear to

be of little value in helping select documents matching a user need are excluded from

the vocabulary entirely. These words are called stop words.

Stemming: Stemming is the process of reducing inflected (or sometimes derived) words to their stem, base, or root form — generally a written word form. For example, if we were to stem the following words: "Stems," "Stemming," "Stemmed," "and Stemtization," the result would be a single word, "stem."

Lemmatization: A slight variant of stemming is lemmatization. The major difference between these is that stemming can often create non-existent words, whereas lemmas are actual words. So, your root stem, meaning the word you end up with, is not something you can look up in a dictionary, but you can look up a lemma.

Examples of Lemmatization are that "run" is a base form for words like "running" or "ran" or that the word "better" and "good" are in the same lemma, so they are considered the same.

Bag of Words

After the initial preprocessing phase, we need to transform the text into a meaningful vector (or array) of numbers. The bag-of-words is a representation of text that describes the occurrence of words within a document. It involves two things:

•A vocabulary of known words.

•A measure of the presence of known words.

Why is it called a “bag” of words?

That is because any information about the order or structure of words in the document is discarded, and the model is only concerned with whether the known words occur in the document, not where they occur in the document.

The intuition behind the Bag of Words is that documents are similar if they have identical content. Also, we can learn something about the meaning of the document from its content alone.

For example, if our dictionary contains the words {Learning, is, the, not, great}, and we want

to vectorize the text “Learning is great,” we would have the following vector: (1, 1, 0, 0, 1).

TF-IDF Approach

A problem with the Bag of Words approach is that highly frequent words start to dominate in the document (e.g., larger score) but may not contain as much “informational content.” Also, it will give more weight to longer documents than shorter documents.

One approach is to rescale the frequency of words by how often they appear in all documents so that the scores for frequent words like “the” that are also frequent across all documents are penalized. This approach to scoring is called Term Frequency-Inverse Document Frequency, or TF-IDF for short, where:

Term Frequency: is a scoring of the frequency of the word in the current document. TF = (Number of times term t appears in a document)/(Number of terms in the document)

Inverse Document Frequency: is a scoring of how rare the word is across documents. IDF = 1+log(N/n), where, N is the number of documents and n is the number of documents a term t has appeared in.

Tf-IDF weight is a weight often used in information retrieval and text mining. This weight is a statistical measure used to evaluate how important a word is to a document in a collection or corpus.

Cosine Similarity

TF-IDF is a transformation applied to texts to get two real-valued vectors in vector space. We can then obtain the Cosine similarity of any pair of vectors by taking their dot product and dividing that by the product of their norms. That yields the cosine of the angle between the vectors. Cosine similarity is a measure of similarity between two non-zero vectors. Using this formula, we can find out the similarity between any two documents d1 and d2.

Cosine Similarity (d1, d2) = Dot product(d1, d2) / ||d1|| * ||d2|| Where d1,d2 are two non-zero vectors.

Importing the necessary libraries

```
import nltk
import numpy as np
import random
import string # to process standard python strings
```

Corpus

For our example, we will be using the Wikipedia page for chatbots as our corpus. Copy the contents from the page and place them in a text file named ‘chatbot.txt.’ However, you can use any corpus of your choice.

Reading in the data

We will read in the corpus.txt file and convert the entire corpus into a list of sentences and a list of words for further pre-processing.

f=open('chatbot.txt','r',errors = 'ignore')
raw=f.read()
raw=raw.lower()# converts to lowercase
nltk.download('punkt') # first-time use only
nltk.download('wordnet') # first-time use only
sent_tokens = nltk.sent_tokenize(raw)# converts to list of sentences
word_tokens = nltk.word_tokenize(raw)# converts to list of words

Let see an example of the sent_tokens and the word_tokens
>>> sent_tokens[:2]
['a chatbot (also known as a talkbot, chatterbot, bot, im bot, interactive agent, or artificial conversational entity) is a computer program or an artificial intelligence which conducts a conversation via auditory or textual methods.', 'such programs are often designed to convincingly simulate how a human would behave as a conversational partner, thereby passing the turing test.']
>>> word_tokens[:2]
['a', 'chatbot', '(', 'also', 'known']

Pre-processing the raw text
We shall now define a function called LemTokens which will take as input the tokens and return normalized tokens.
lemmer = nltk.stem.WordNetLemmatizer()
#WordNet is a semantically-oriented dictionary of English included in NLTK.
def LemTokens(tokens):
return [lemmer.lemmatize(token) for token in tokens]
remove_punct_dict = dict((ord(punct), None) for punct in string.punctuation)
def LemNormalize(text):
return LemTokens(nltk.word_tokenize(text.lower().translate(remove_punct_dict)))

Keyword matching
Next, we shall define a function for a greeting by the bot, i.e., if a user's input is a greeting, the bot shall return a greeting response.ELIZA uses a simple keyword matching for greetings. We will utilize the same concept here.

GREETING_INPUTS = ("hello", "hi", "greetings", "sup", "what's up","hey",)

```
GREETING_RESPONSES = ["hi", "hey", "*nods*", "hi there", "hello", "I am glad! You are talking to me"]
def greeting(sentence):
for word in sentence.split():
if word.lower() in GREETING_INPUTS:
return random.choice(GREETING_RESPONSES)
```

Generating Response

To generate a response from our bot for input questions, the concept of document similarity will be used. So we begin by importing the necessary modules.

● From the scikit learn library, import the TFidf vectorizer to convert a collection of raw documents to a matrix of TF-IDF features. from sklearn.feature_extraction.text import TfidfVectorizer

● Also, import cosine similarity module from scikit learn library from sklearn.metrics.pairwise import cosine_similarity

This will be used to find the similarity between words entered by the user and the words in the corpus. This is the simplest possible implementation of a chatbot. We define a function response that searches the user's utterance for one or more general keywords and returns one of several possible responses. If it doesn't find the input matching any of the keywords, it returns a response:" I am sorry! I don't understand you."

```
def response(user_response):
robo_response=''
sent_tokens.append(user_response)
TfidfVec = TfidfVectorizer(tokenizer=LemNormalize, stop_words='english')
tfidf = TfidfVec.fit_transform(sent_tokens)
vals = cosine_similarity(tfidf[-1], tfidf)
idx=vals.argsort()[0][-2]
flat = vals.flatten()
flat.sort()
req_tfidf = flat[-2]
if(req_tfidf==0):
robo_response=robo_response+"I am sorry! I don't understand you"
```

```
return robo_response
else:
robo_response = robo_response+sent_tokens[idx]
return robo_response
```

Finally, we will feed the lines that we want our bot to say while starting and ending a conversation, depending upon the user's input.

```
flag=True
print("ROBO: My name is Robo. I will answer your queries about Chatbots. If you want to
exit, type Bye!")
while(flag==True):
user_response = input()
user_response=user_response.lower()
if(user_response!='bye'):
if(user_response=='thanks' or user_response=='thank you' ):
flag=False
print("ROBO: You are welcome..")
else:
if(greeting(user_response)!=None):
print("ROBO: "+greeting(user_response))
else:
print("ROBO: ",end="")
print(response(user_response))
sent_tokens.remove(user_response)
else:
flag=False
print("ROBO: Bye! take care..")
```

Code

```
import io
import random
import string # to process standard python strings
```

```
import warnings
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
import warnings
warnings.filterwarnings('ignore')
import nltk
from nltk.stem import WordNetLemmatizer
nltk.download('popular', quiet=True) # for downloading packages
with open('chatbot.txt','r', encoding='utf8', errors ='ignore') as fin:
raw = fin.read().lower()
#TOkenisation
sent_tokens = nltk.sent_tokenize(raw)# converts to list of sentences
word_tokens = nltk.word_tokenize(raw)# converts to list of words
# Preprocessing
lemmer = WordNetLemmatizer()
def LemTokens(tokens):
return [lemmer.lemmatize(token) for token in tokens]
remove_punct_dict = dict((ord(punct), None) for punct in string.punctuation)
def LemNormalize(text):
return LemTokens(nltk.word_tokenize(text.lower().translate(remove_punct_dict)))
# Keyword Matching
GREETING_INPUTS = ("hello", "hi", "greetings", "sup", "what's up","hey",)
GREETING_RESPONSES = ["hi", "hey", "*nods*", "hi there", "hello", "I am glad! You are
talking to me"]
def greeting(sentence):
"""If user's input is a greeting, return a greeting response"""
for word in sentence.split():
if word.lower() in GREETING_INPUTS:
return random.choice(GREETING_RESPONSES)
# Generating response
def response(user_response):
robo_response=''
sent_tokens.append(user_response)
```

```
TfidfVec = TfidfVectorizer(tokenizer=LemNormalize, stop_words='english')
tfidf = TfidfVec.fit_transform(sent_tokens)
vals = cosine_similarity(tfidf[-1], tfidf
idx=vals.argsort()[0][-2]
flat = vals.flatten()
flat.sort()
req_tfidf = flat[-2]
if(req_tfidf==0):
robo_response=robo_response+"I am sorry! I don't understand you"
return robo_response
else:
robo_response = robo_response+sent_tokens[idx]
return robo_response
flag=True
print("ROBO: My name is Robo. I will answer your queries about Chatbots. If you want to
exit, type Bye!")
while(flag==True):
user_response = input("You: ")
user_response=user_response.lower()
if(user_response!='bye'):
if(user_response=='thanks' or user_response=='thank you' ):
flag=False
print("ROBO: You are welcome..")
else:
if(greeting(user_response)!=None):
print("ROBO: "+greeting(user_response))
else:
print("ROBO: ",end="")
print(response(user_response))
sent_tokens.remove(user_response)
else:
flag=False
print("ROBO: Bye! take care..")
```

Data:
(chatbot.txt) [source: wikipedia - robots]

A chatbot (also known as a talkbot, chatterbot, Bot, IM bot, interactive agent, or Artificial Conversational Entity) is a computer program or an artificial intelligence which conducts a conversation via auditory or textual methods. Such programs are often designed to convincingly simulate how a human would behave as a conversational partner, there passing the Turing test. Chatbots are typically used in dialog systems for various practical purposes including customer service or information acquisition. Some chatterbots use sophisticated natural language processing systems, but many simpler systems scan for keywords within the input, then pull a reply with the most matching keywords, or the most similar wording pattern, from a database.

The term "ChatterBot" was originally coined by Michael Mauldin (creator of the first Verbot, Julia) in 1994 to describe these conversational programs.Today, most chatbots are either accessed via virtual assistants such as Google Assistant and Amazon Alexa, viamessaging apps such as Facebook Messenger or WeChat, or via individual organizations'

A chatbot (also known as a talkbot, chatterbot, Bot, IM bot, interactive agent, or Artificial Conversational Entity) is a computer program or an artificial intelligence which conducts a conversation via auditory or textual methods. Such programs are often designed to convincingly simulate how a human would behave as a conversational partner, thereby passing the Turing test. Chatbots are typically used in dialog systems for various practical purposes including customer service or information acquisition. Some chatterbots use sophisticated natural language processing systems, but many simpler systems scan for keywords within the input, then pull a reply with the most matching keywords, or the most similar wording pattern, from a database.

The term "ChatterBot" was originally coined by Michael Mauldin (creator of the first Verbot, Julia) in 1994 to describe these conversational programs.Today, most chatbots are either accessed via virtual assistants such as Google Assistant and Amazon Alexa, via messaging apps such as Facebook Messenger or WeChat, or via individual organizations' apps and websites. Chatbots can be classified into usage categories such as conversational commerce (e-commerce via chat), analytics, communication, customer support, design, developer tools, education, entertainment, finance, food, games, health, HR, marketing, news, personal, productivity, shopping, social, sports, travel and utilities.

Background

In 1950, Alan Turing's famous article "Computing Machinery and Intelligence" was published, which proposed what is now called the Turing test as a criterion of intelligence.

This criterion depends on the ability of a computer program to impersonate a human in a real-time written conversation with a human judge, sufficiently well that the judge is unable to distinguish reliably on the basis of the conversational content alone between the program and a real human. The notoriety of Turing's proposed test stimulated great interest in Joseph Weizenbaum's program ELIZA, published in 1966, which seemed to be able to fool users into believing that they were conversing with a real human. However Weizenbaum himself did not claim that ELIZA was genuinely intelligent, and the

Introduction to his paper presented it more as a debunking exercise: [In] artificial intelligence ... machines are made to behave in wondrous ways, often sufficient to dazzle even the most experienced observer. But once a particular program is unmasked, once its inner workings are explained ... its magic crumbles away; it stands revealed as a mere collection of procedures ... The observer says to himself "I could have written that". With that thought he moves the program in question from the shelf marked "intelligent", to that reserved for curios ... The object of this paper is to cause just such a

re-evaluation of the program about to be "explained". Few programs ever needed it more. ELIZA's key method of operation (copied by chatbot designers ever since) involves the recognition of cue words or phrases in the input, and the output of corresponding pre-prepared or pre-programmed responses that can move the conversation forward in an apparently meaningful way (e.g. by responding to any input that contains the word 'MOTHER' with 'TELL ME MORE ABOUT YOUR FAMILY').Thus an illusion of understanding is generated, even though the processing involved has been merely superficial. ELIZA showed that such an illusion is surprisingly easy to generate, because

human judges are so ready to give the benefit of the doubt when conversational responses are capable of being interpreted as "intelligent".

Interface designers have come to appreciate that humans' readiness to interpret computer output as genuinely conversational even when it is actually based on rather simple pattern-matching can be exploited for useful purposes. Most people prefer to engage with programs that are human-like, and this gives chatbot-style techniques a potentially useful role in interactive systems that need to elicit information from users, as long as that information is relatively straightforward and falls into predictable categories. Thus, for example, online help systems can usefully employ chatbot techniques to identify the area of help that users require, potentially providing a "friendlier" interface than a more formal search or menu system. This sort of usage holds the prospect of moving chatbot technology from Weizenbaum's "shelf ... reserved for curios" to that marked "genuinely useful computational methods".

Development

The classic historic early chatbots are ELIZA (1966) and PARRY (1972).More recent notable programs include A.L.I.C.E., Jabberwacky and D.U.D.E (Agence Nationale de la Recherche and CNRS 2006). While ELIZA and PARRY were used exclusively to simulate typed conversation, many chatbots now include functional features such as games and web searching abilities. In 1984, a book called The Policeman's Beard is Half Constructed was published, allegedly written by the chatbot Racter (though the program as released would not have been capable of doing so).

One pertinent field of AI research is natural language processing. Usually, weak AI fields employ specialized software or programming languages created specifically for the narrow function required. For example, A.L.I.C.E. uses a markup language called AIML, which is specific to its function as a conversational agent, and has since been adopted by various other developers of, so called, Alicebots. Nevertheless, A.L.I.C.E. is still purely based on pattern matching techniques without any reasoning capabilities, the same technique ELIZA was using back in 1966. This is not strong AI, which would require sapience and logical reasoning abilities.

Jabberwacky learns new responses and context based on real-time user interactions, rather than being driven from a static database. Some more recent chatbots also combine real-time learning with evolutionary algorithms that optimise their ability to communicate based on each conversation held. Still, there is currently no general purpose conversational artificial intelligence, and some software developers focus on the practical aspect, information retrieval.

Chatbot competitions focus on the Turing test or more specific goals. Two such annual contests are the Loebner Prize and The Chatterbox Challenge (offline since 2015, materials can still be found from web archives).

According to Forrester (2015), AI will replace 16 percent of American jobs by the end of the decade.Chatbots have been used in applications such as customer service, sales and product education. However, a study conducted by Narrative Science in 2015 found that 80 percent of their respondents believe AI improves worker performance and creates jobs.[citation needed]

Application

Aeromexico airline chatbot running on Facebook Messenger, March 2018

Messaging apps

Many companies' chatbots run on messaging apps like Facebook Messenger (since 2016), WeChat (since 2013),WhatsApp, LiveChat, Kik, Slack, Line, Telegram, or simply via SMS. They are used for B2C customer service, sales and marketing.

In 2016, Facebook Messenger allowed developers to place chatbots on their platform. There were 30,000 bots created for Messenger in the first six months, rising to 100,000 by September 2017.

Since September 2017, this has also been as part of a pilot program on WhatsApp. Airlines KLM and Aeromexico both announced their participation in the testing;both airlines had previously launched customer services on the Facebook Messenger platform. The bots usually appear as one of the user's contacts, but can sometimes act as participants in a group chat.

Many banks and insurers, media and e-commerce companies, airlines and hotel chains, retailers, health care providers, government entities and restaurant chains have used chatbots to answer simple questions, increase customer engagement,for promotion, and to offer additional ways to order from them.

A 2017 study showed 4% of companies used chatbots.According to a 2016 study, 80% of businesses said they intended to have one by 2020.

As part of company apps and websites Previous generations of chatbots were present on company websites, e.g. Ask Jenn from

Alaska Airlines which debuted in 2008 or Expedia's virtual customer service agent which launched in 2011.The newer generation of chatbots includes IBM Watson-powered "Rocky", introduced in February 2017 by the New York City-based e-commerce company Rare Carat to provide information to prospective diamond buyers. Company internal platforms

Other companies explore ways they can use chatbots internally, for example for Customer Support, Human Resources, or even in Internet-of-Things (IoT) projects. Overstock.com, for one, has reportedly launched a chatbot named Mila to automate certain simple yet time-consuming processes when requesting for a sick leave.Other large companies such as Lloyds Banking Group, Royal Bank of Scotland, Renault and Citroen are now using automated online assistants instead of call centres with humans to provide a first point of contact. A SaaS chatbot business ecosystem has been steadily growing since the F8 Conference when Zuckerberg unveiled that Messenger would allow chatbots into the app.

Toys

Chatbots have also been incorporated into devices not primarily meant for computing such as toys.

Hello Barbie is an Internet-connected version of the doll that uses a chatbot provided by the company ToyTalk,which previously used the chatbot for a range of smartphone-based characters for children.These characters' behaviors are constrained by a set of rules that in effect emulate a particular character and produce a storyline.

IBM's Watson computer has been used as the basis for chatbot-based educational toys for companies such as CogniToys intended to interact with children for educational purposes.

Chatbot creation

The process of creating a chatbot follows a pattern similar to the development of a web page or a mobile app. It can be divided into Design, Building, Analytics and Maintenance.

Design

The chatbot design is the process that defines the interaction between the user and the chatbot.The chatbot designer will define the chatbot personality, the questions that will be asked to the users, and the overall interaction.It can be viewed as a subset of the conversational design. In order to speed up this process, designers can use dedicated chatbot design tools, that allow for immediate preview, team collaboration and video export.An important part of the chatbot design is also centered around user testing. User testing can be performed following the same principles that guide the user testing of
graphical interfaces.

Building

The process of building a chatbot can be divided into two main tasks: understanding the user's intent and producing the correct answer. The first task involves understanding the user input. In order to properly understand a user input in a free text form, a Natural
Language Processing Engine can be used.The second task may involve different approaches depending on the type of the response that the chatbot will generate.

Analytics

The usage of the chatbot can be monitored in order to spot potential flaws or problems. It can also provide useful insights that can improve the final user experience.

Maintenance

To keep chatbots up to speed with changing company products and services, traditional chatbot development platforms require ongoing maintenance. This can either be in the form of an ongoing service provider or for larger enterprises in the form of an in-house chatbot training team.To eliminate these costs, some startups are experimenting with Artificial Intelligence to develop self-learning chatbots, particularly in Customer Service applications.

Chatbot development platforms

The process of building, testing and deploying chatbots can be done on cloud based chatbot development platforms offered by cloud Platform as a Service (PaaS) providers such as Yekaliva, Oracle Cloud Platform, SnatchBot and IBM Watson.These cloud platforms provide Natural Language Processing, Artificial Intelligence and Mobile Backend as a Service for chatbot development.

APIs

There are many APIs available for building your own chatbots, such as AARC.

Malicious use

Malicious chatbots are frequently used to fill chat rooms with spam and advertisements, by mimicking human behaviour and conversations or to entice people into revealing personal information, such as bank account numbers. They are commonly found on Yahoo! Messenger, Windows Live Messenger, AOL Instant Messenger and other instant messaging protocols. There has also been a published report of a chatbot used in a fake personal ad on a dating service's website.

Steps to run:

- Make sure chatbot.py and chatbot.txt are in the same directory.
- Run chatbot.py

$ python chatbot.py