EXPERIMENT - 1

Aim: Perform pre-processing of text on any dataset

Theory:
We performed a series of steps under each component based on the general outline above.

1. Remove HTML tags
2. Remove extra whitespaces
3. Convert accented characters to ASCII characters
4. Expand contractions
5. Remove special characters
6. Lowercase all texts
7. Convert number words to numeric form
8. Remove numbers
9. Remove stopwords
10. Lemmatization
11. Tokenisation
12. Stemming
13. Normalisation
14. POS Tagging

CODE:

```
from indicnlp.normalize.indic_normalize import IndicNormalizerFactory

from idatasets import load_devdas
from nltk.stem import PorterStemmer
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer
from nltk.stem import PorterStemmer
import spacy
import pandas as pd

import stanfordnlp
```

```
import requests
from bs4 import BeautifulSoup
import unicodedata
import contractions
from word2number import w2n
import re
import stanza

from inltk.inltk import setup
from inltk.inltk import tokenize

URL = "http://www.values.com/inspirational-quotes"
r = requests.get(URL)
text = r.content
```

Remove HTML Tags

If the reviews or texts are web-scraped, chances are they will contain some HTML tags. Since these tags are not useful for our NLP tasks, it is better to remove them.

```
def remove_html(text):
    soup = BeautifulSoup(text, "lxml")
    text = soup.get_text()
    return str(text)
```

```
output1 = remove_html(text)
output1
```

```
\n\n\n\n\n\nInspirational Quotes - Motivational Quotes - | The Foundation for a Better Life\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n window.dataLayer = window.dataLayer || [];\n function
```

```
gtag(){dataLayer.push(arguments);}\n gtag('js', new Date());\n gtag('config', 'UA-1179606-29');\n \n\n window.fbAsyncInit = function() {\n FB.init({\n appId : '483774921971842',\n autoLogAppEvents :...
```

Removing extra whitespaces and tabs

Extra whitespaces and tabs do not add any information to text processing.

```
def remove_whitespace(text):
    text = " ".join(text.split())
    return text
```

```
output2 = remove_whitespace(output1)
output2
```

```
Inspirational Quotes - Motivational Quotes - | The Foundation for a Better Life window.dataLayer = window.dataLayer || []; function gtag(){dataLayer.push(arguments);} gtag('js', new Date()); gtag('config', 'UA-1179606-29'); window.fbAsyncInit = function() { FB.init({ appId : '483774921971842', autoLogAppEvents : true, xfbml : true, version : 'v6.0' }); }; (function(w,d,s,l,i){w[l]=w[l]||[];w[l].push({'gtm.start': new Date().getTime()...
```

Convert Accented Characters

Words with accent marks like “latté” and “café” can be converted and standardized to just “latte” and “cafe”, else our NLP model will treat “latté” and “latte” as different words even though they are referring to same thing.

```
def accented_to_ascii(text):
    try:
        text = unicode(text, "utf-8")
```

```
    except (TypeError, NameError):  # unicode is a default on python 3
        pass
    text = unicodedata.normalize("NFD", text)
    text = text.encode("ascii", "ignore")
    text = text.decode("utf-8")
    return str(text)
```

```
output3 = accented_to_ascii(output2)
output3
```

Expand Contractions

Contractions are shortened words, e.g., don't and can't. Expanding such words to "do not" and "can not" helps to standardize text.

We use the contractions module to expand the contractions.

```
def expand_contractions(text):
    expanded_words = []
    for word in text.split():
        expanded_words.append(contractions.fix(word))
    expanded_text = " ".join(expanded_words)
    return expanded_text
```

```
output4 = expand_contractions(output3)
output4
```

```
Inspirational Quotes - Motivational Quotes - | The Foundation for a Better Life window.dataLayer = window.dataLayer || []; function gtag(){dataLayer.push(arguments);} gtag('js', new Date()); gtag('config', 'UA-1179606-29'); window.fbAsyncInit = function() { FB.init({ appId : '483774921971842', autoLogAppEvents : true, xfbml : true, version : 'v6.0' }); }; (function(w,d,s,l,i){w[l]=w[l]||[];w[l].push({'gtm.start': new Date().getTime(),...
```

Removing Special Characters

Special characters, as you know, are non-alphanumeric characters. These characters are most often found in comments, references, currency numbers etc. These characters add no value to text-understanding and induce noise into algorithms.

```
def remove_special(text):
    text = text.split()
    text = " ".join(x for x in text if not x.isalnum())
    text = text.split()
    special_char_list = ["$", "@", "#", "&", "%"]
    text = " ".join([k for k in text if k not in special_char_list])
    text = " ".join(text.split())
    return text
```

```
output5 = remove_special(output4)
output5
```

```
- - | window.dataLayer = window.dataLayer || []; gtag(){dataLayer.push(arguments);} gtag('js', Date()); gtag('config', 'UA-1179606-29'); window.fbAsyncInit = function() { FB.init({ : '483774921971842', : true, : true, : 'v6.0' }); }; (function(w,d,s,l,i){w[l]=w[l]||[];w[l].push({'gtm.start': Date().getTime(),event:'gtm.js'});var f=d.getElementsByTagName(s)[0], j=d.createElement(s),dl=l!='dataLayer'?'&l='+l:'';j.async=true;j.src= …
```

Lowercase

Changing case to lower can be achieved by using lower function.

```
def text_to_lowercase(text):
    text = text.lower()
    return text
```

```
output6 = text_to_lowercase(output5)
output6
```

Treatment for Numbers

There are two steps in our treatment of numbers.

One of the steps involve the conversion of number words to numeric form, e.g., seven to 7, to standardize text. To do this, we use the word2number module. The other step is to remove numbers.

```
def number_word_to_numeric(text):
    text = text.split()
    output = ""
    for i in text:
        try:
            res = w2n.word_to_num(i)
        except:
            res = i
        output += str(res) + " "
    output = output.rstrip()
    return output


def remove_number(text):
    res = " ".join([i for i in text if not i.isdigit()])
    return res
```

Remove stopwords

Stopwords are very common words. Words like “we” and “are” probably do not help at all in NLP tasks such as sentiment analysis or text classifications. Hence, we can remove stopwords to save computing time and efforts in processing large volumes of text.

```
def remove_stop_words(text):
```

```
stop = open("stopwords.txt")

stop_words = []

for x in stop:
    stop_words.append(x)


stop_words = list(set(stop_words))

word_tokens = word_tokenize(text)
filtered_sentence = []

for w in word_tokens:
    if w not in stop_words:
        filtered_sentence.append(w)

filtered_sentence = " ".join(filtered_sentence)
return filtered_sentence
```

Lemmatization:

Like stemming, lemmatization also converts a word to its root form. The only difference is that lemmatization ensures that the root word belongs to the language. We will get valid words if we use lemmatization. In NLTK, we use the WordNetLemmatizer to get the lemmas of words. We also need to provide a context for the lemmatization.

```
def lemmatization(text):
    nlp = stanza.Pipeline(lang="en", processors="tokenize, pos, lemma")
    doc = nlp(text)
    parsed_text = {"word": [], "lemma": []}
    for sent in doc.sentences:
        for wrd in sent.words:
            parsed_text["word"].append(wrd.text)
            parsed_text["lemma"].append(wrd.lemma)
```

```
return pd.DataFrame(parsed_text)
```

```
INFO:stanza:Checking for updates to resources.json in case models have been updated.  Note: this behavior can be turned off with download_method=None or download_method=DownloadMethod.REUSE_RESOURCES

Downloading https://raw.githubusercontent.com/stanfordnlp/stanza-resources/main/resources_1.4.1.json: 193k/? [00:00<00:00, 5.55MB/s]
INFO:stanza:Loading these models for language: en (English):
========================
| Processor | Package  |
------------------------
| tokenize  | combined |
| pos       | combined |
| lemma     | combined |
========================

INFO:stanza:Use device: cpu
INFO:stanza:Loading: tokenize
INFO:stanza:Loading: pos
INFO:stanza:Loading: lemma
INFO:stanza:Done loading processors!


word
lemma
0
Inspirational
inspirational
1
Quotes
quote
2
-
-
3
Motivational
Motivational
```

```
4
Quotes
quote
...
...
...
915
token
token
916
;
;
917
}
}
918
No
no
919
thanks
thanks
920 rows × 2 columns
```

Tokenization

In natural language processing, tokenization is the text preprocessing task of breaking up text into smaller components of text (known as tokens).

```
def tokenization(text):
    tokenized_text = tokenize(text, "en")
    return tokenized_text
```

```
['This', 'is', 'a', 'sample', 'sentence', 'for', 'tokenization']
```

Stemming:

Stemming is the process of getting the root form of a word. Stem or root is the part to which inflectional affixes (-ed, -ize, -de, -s, etc.) are added. The stem of a word is created by removing the prefix or suffix of a word. So, stemming a word may not result in actual words.

```
def stemming(text):
    ps = PorterStemmer()
    text = text.split()
    output = ""
    for i in text:
        res = ps.stem(i)
        output += str(res) + " "
    return output
```

```
from nltk.stem import PorterStemmer

def stemming(text):
  ps = PorterStemmer()
  text = text.split()
  output = ""
  for i in text:
    res = ps.stem(i)
    output += (str(res) + " ")
  return output

[69] stemming("The cats are playing with each other in the garden.")

'the cat are play with each other in the garden. '
```

Normalization

Usually, text normalisation starts with tokenizing the text, which our longer corpus is now to be split into chunks of words, which the tokenizer class from NLTK can do. Post that, we need to lower case each word of our corpus, converting numbers to the words and last with contraction replacement.

```
def text_normalization(text):
    factory = IndicNormalizerFactory()
    normalizer = factory.get_normalizer("en")
    text = normalizer.normalize(text)
    return text
```

Part-of-Speech Tagging

In natural language processing, part-of-speech tagging is the process of assigning a part of speech to every word in a string. Using the part of speech can improve the results of lemmatization.

```
def pos_tagging(text):
    nlp = stanza.Pipeline(lang="en", processors="tokenize, pos, lemma")
    doc = nlp(text)
    parsed_text = {"word": [], "upos": [], "xpos": []}
    for sent in doc.sentences:
        for wrd in sent.words:
            parsed_text["word"].append(wrd.text)
            parsed_text["upos"].append(wrd.upos)
            parsed_text["xpos"].append(wrd.xpos)
    return pd.DataFrame(parsed_text)
```

[('My', PRP$'), ('name', 'NN'), ('is', 'VBZ'), (‘Junaid’, 'NNP'), (‘.’, ‘.’)]

Conclusion:

We have successfully performed text pre-processing tasks on a given piece of English text.

EXPERIMENT - 2

Aim: Generate bigrams and trigrams from a given corpus and calculate probability of a sentence.

Theory:
Probability of a sentence can be calculated by the probability of sequence of words occurring in it. We can use Markov assumption that the probability of a word in a sentence depends on the probability of the word occurring just before it. Such a model is called first order Markov model or the bigram model.

$$P(W_n \mid W_{n-1}) = P(W_{n-1},W_n)/P(W_{n-1})$$

Here, Wn refers to the word token corresponding to the nth word in a sequence.
A combination of words forms a sentence. However, such a formation is meaningful only when the words are arranged in some order.
Eg: Sit I car in the
Such a sentence is not grammatically acceptable. However some perfectly grammatical sentences can be nonsensical too!
Eg: Colorless green ideas sleep furiously
One easy way to handle such unacceptable sentences is by assigning probabilities to the strings of words i.e, how likely the sentence is.

Probability of a sentence
If we consider each word occurring in its correct location as an independent event, the probability of the sentences is : P(w(1), w(2)..., w(n-1), w(n))

Using chain rule:
=P(w(1)) * P(w(2) | w(1)) * P(w(3) | w(1)w(2)) ... P(w(n) | w(1)w(2)…w(n-1))

Bigrams
We can avoid this very long calculation by approximating that the probability of a given word depends only on the probability of its previous words. This assumption is called Markov assumption and such a model is called Markov model- bigrams.

Bigrams can be generalized to the n-gram which looks at (n-1) words in the past. A bigram is a first-order Markov model.
Therefore ,
P(w(1), w(2)..., w(n-1), w(n))= P(w(2)|w(1)) P(w(3)|w(2)) …. P(w(n)|w(n-1))

We use (eos) tag to mark the beginning and end of a sentence.
A bigram table for a given corpus can be generated and used as a lookup table for calculating probability of sentences.

Eg: Corpus – (eos) You book a flight (eos) I read a book (eos) You read (eos)

Code & Output:

```
corpus = "You book a flight. I read a book. You read."
import nltk
nltk.download('punkt')
```

```
def preprocess(d):
    d=d.lower()
    d="eos "+ d
    d=d.replace("."," eos")
    return d
d=preprocess(corpus)
print("Preprocessed Data corpus = \n",d)
```

```
Preprocessed Data corpus =
 eos you book a flight eos i read a book eos you read eos
```

```
from nltk import word_tokenize
def generate_tokens(d):
    tokens = word_tokenize(d)
    return tokens
```

```
tokens=generate_tokens(d)
distinct_tokens = list(set(sorted(tokens)))
print("Tokens in the corpus = \n",distinct_tokens)
```

```
Tokens in the corpus =
 ['book', 'a', 'flight', 'you', 'eos', 'read', 'i']
```

```
def generate_tokens_freq(tokens):
    dct={}
    for i in tokens:
        dct[i]=0
    for i in tokens:
        dct[i]+=1
    return dct
dct=generate_tokens_freq(tokens)
print("Frequency of each tokens = ")
for i in dct.items():
    print(i[0],"\t:" , i[1])
```

```
Frequency of each tokens =
eos     : 4
you     : 2
book    : 2
a       : 2
flight  : 1
i       : 1
read    : 2
```

```
def generate_ngrams(tokens,k):
    l=[]
```

```
    i=0
    while(i<len(tokens)):
        l.append(tokens[i:i+k])
        i=i+1
    l=l[:-1]
    return l
bigram = generate_ngrams(tokens,2)
print("N-grams generated (Here n is 2) = ")
for i in bigram:
    print(i)
```

```
N-grams generated (Here n is 2) =
['eos', 'you']
['you', 'book']
['book', 'a']
['a', 'flight']
['flight', 'eos']
['eos', 'i']
['i', 'read']
['read', 'a']
['a', 'book']
['book', 'eos']
['eos', 'you']
['you', 'read']
['read', 'eos']
```

```
def generate_ngram_freq(bigram):
    dct1={}
    for i in bigram:
        st=" ".join(i)
        dct1[st]=0
```

```
    for i in bigram:
        st=" ".join(i)
        dct1[st]+=1
    return dct1
dct1=generate_ngram_freq(bigram)
print("Frequency of n-grams = ")
for i in dct1.items():
    print(i[0], ":", i[1])
```

```
Frequency of n-grams =
eos you : 2
you book : 1
book a : 1
a flight : 1
flight eos : 1
eos i : 1
i read : 1
read a : 1
a book : 1
book eos : 1
you read : 1
read eos : 1
```

```
def find1(s,dct1):
    try:
        return dct1[s]
    except:
        return 0
def print_probability_table(distinct_tokens,dct,dct1):
    n=len(distinct_tokens)
    l=[[]*n for i in range(n)]
    for i in range(n):
        denominator = dct[distinct_tokens[i]]
        for j in range(n):
```

```
        numerator = find1(distinct_tokens[i]+" "+distinct_tokens[j],dct1)
        l[i].append(float("{:.3f}".format(numerator/denominator)))
    return l


print("Probability table = \n")
probability_table=print_probability_table(distinct_tokens,dct,dct1)
n=len(distinct_tokens)
print("\t", end="")
for i in range(n):
    print(distinct_tokens[i],end="\t")
print("\n")
for i in range(n):
    print(distinct_tokens[i],end="\t")
    for j in range(n):
        print(probability_table[i][j],end="\t")
    print("\n")
```

```
Probability table =

        book    a       flight  you     eos     read    i

book    0.0     0.5     0.0     0.0     0.5     0.0     0.0

a       0.5     0.0     0.5     0.0     0.0     0.0     0.0

flight  0.0     0.0     0.0     0.0     1.0     0.0     0.0

you     0.5     0.0     0.0     0.0     0.0     0.5     0.0

eos     0.0     0.0     0.0     0.5     0.0     0.0     0.25

read    0.0     0.5     0.0     0.0     0.5     0.0     0.0

i       0.0     0.0     0.0     0.0     0.0     1.0     0.0
```

```
input_text = "You read a book."

p = preprocess(input_text)
print("Preprocessed Text = \n",p,"\n")
t = generate_tokens(p)
print("Tokens Generated = \n",t,"\n")
n = generate_ngrams(t,2)
print("N-grams Generated = \n= ",n)
```

```
Preprocessed Text =
 eos you read a book eos

Tokens Generated =
 ['eos', 'you', 'read', 'a', 'book', 'eos']

N-grams Generated =
=  [['eos', 'you'], ['you', 'read'], ['read', 'a'], ['a', 'book'], ['book', 'eos']]
```

```
for i in n:
    print('"{}"'.format(' '.join(i)), end=", ")
print("\n\n"+'\033[1m'+"Calculate bigram probability"+'\033[0m')
s=1
dct2={}
for i in n:
    dct2[" ".join(i)]=0

for i in n:
    k=distinct_tokens.index(i[0])
    m=distinct_tokens.index(i[1])
    dct2[" ".join(i)]=probability_table[k][m]
    print("P('{}')\t=  ".format(' '.join(i)),probability_table[k][m])
```

```
    s*=probability_table[k][m]

print("\n"+'\033[1m'+ "Calculate Probability of the sentence"+'\033[0m')
print(f"P('{input_text}') \n= ",end="")
x=dct2.popitem()
for i in dct2:
    print(f"P('{i}')", end=" * ")
print(f"P('{x[0]}')\n= ", end=")

for i in dct2:
    print(dct2[i], end=" * ")
print(x[1],"\n=",s)

print("\n"+'\033[1m'+f"Probability('{input_text}') = "+"{:.5f}".format(s))
```

```
'eos you', 'you read', 'read a', 'a book', 'book eos',

Calculate bigram probability
P('eos you')    =   0.5
P('you read')   =   0.5
P('read a')     =   0.5
P('a book')     =   0.5
P('book eos') =   0.5

Calculate Probability of the sentence
P('You read a book.')
= P('eos you') * P('you read') * P('read a') * P('a book') * P('book eos')
= 0.5 * 0.5 * 0.5 * 0.5 * 0.5
= 0.03125

Probability('You read a book.') = 0.03125
```

```
trigram = generate_ngrams(tokens,3)
```

```
print("N-grams generated (Here n is 3) = ")
for i in trigram:
    print(i)
```

```
N-grams generated (Here n is 3) =
['eos', 'you', 'book']
['you', 'book', 'a']
['book', 'a', 'flight']
['a', 'flight', 'eos']
['flight', 'eos', 'i']
['eos', 'i', 'read']
['i', 'read', 'a']
['read', 'a', 'book']
['a', 'book', 'eos']
['book', 'eos', 'you']
['eos', 'you', 'read']
['you', 'read', 'eos']
['read', 'eos']
```

```
dct2=generate_ngram_freq(trigram)
print("Frequency of n-grams = ")
for i in dct2.items():
    print(i[0], ":", i[1])
```

```
Frequency of n-grams =
eos you book : 1
you book a : 1
book a flight : 1
a flight eos : 1
flight eos i : 1
eos i read : 1
i read a : 1
```

```
read a book : 1
a book eos : 1
book eos you : 1
eos you read : 1
you read eos : 1
read eos : 1
```

Conclusion: In this experiment, we have implemented Bigrams and Trigrams using python.

EXPERIMENT - 3

Aim: To perform

1) Chunking for verb/prepositional phrases using NLTK and Spacy.
2) POS Tagging, Count the POS tags (counter function) and visualization of Frequency distribution of each word in the graph (take a sample from corpus)
3) Named Entity Recognition

Theory:

Named-entity recognition (NER) (also known as entity identification, entity chunking and entity extraction) is a subtask of information extraction that seeks to locate and classify named entity mentioned in unstructured text into pre-defined categories such as person names, organizations, locations, medical codes, time expressions, quantities, monetary values, percentages, etc.

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, it's advisable to use phrases such as "South Africa" as a single word instead of 'South' and 'Africa' separate words.

Chunking works on top of POS tagging, it uses POS-tags as input and provides chunks as output. Similar to POS tags, there are a standard set of Chunk tags like Noun Phrase (NP), Verb Phrase (VP), etc. Chunking is very important when you want to extract information from text such as Locations, Person Names etc. In NLP called Named Entity Extraction.

There are a lot of libraries which gives phrases out-of-box such as Spacy or TextBlob. NLTK just provides a mechanism using regular expressions to generate chunks.

Steps to be performed:

To perform Noun Phrase Chunking, search for chunks corresponding to an individual noun phrase.

In order to create NP chunk, define the chunk grammar using POS tags.

Define this using a single regular expression rule ('"

NP: {<DT>?<JJ>*<NN>} # NP

"')

The rule states that whenever the chunk finds an optional determiner (DT) followed by any number of adjectives (JJ) and then a noun (NN) then the Noun Phrase (NP) chunk should be formed. Then perform chunking using the following:

chunkParser = nltk.RegexpParser(grammar)

tagged = nltk.pos_tag(nltk.word_tokenize(sentence))

Code:

```
from pprint import pprint

import en_core_web_sm

nlp = en_core_web_sm.load()

import nltk

ex = "The bilateral trade between the two countries is expected to rise from existing 27 billion USD to 45 billion USD in the next five years, union minister Piyush Goyal said as he signed the deal with Australian trade minister Dan Tehan."


def preprocess(sent):
    sent = nltk.word_tokenize(sent)
    sent = nltk.pos_tag(sent)
```

```
    return sent


sent = preprocess(ex)
print(sent)
pattern = "NP: {<DT>?<JJ>*<NN>}"
cp = nltk.RegexpParser(pattern)
tree = cp.parse(sent)
for subtree in tree.subtrees():
    print(subtree)

doc = nlp(ex)
pprint([(X.text, X.label_) for X in doc.ents])
tree.draw()
```

Output:

```
[('The', 'DT'), ('bilateral', 'JJ'), ('trade', 'NN'), ('between', 'IN'), ('the', 'DT'), ('two', 'CD'), ('countries', 'NNS'), ('is', 'VBZ'), ('expected', 'VBN'), ('to', 'TO'), ('rise', 'VB'), ('from', 'IN'), ('existing', 'VBG'), ('27', 'CD'), ('billion', 'CD'), ('USD', 'NNP'), ('to', 'TO'), ('45', 'CD'), ('billion', 'CD'), ('USD', 'NNP'), ('in', 'IN'), ('the', 'DT'), ('next', 'JJ'), ('five', 'CD'), ('years', 'NNS'), (',', ','), ('union', 'NN'), ('minister', 'NN'), ('Piyush', 'NNP'), ('Goyal', 'NNP'), ('said', 'VBD'), ('as', 'IN'), ('he', 'PRP'), ('signed', 'VBD'), ('the', 'DT'), ('deal', 'NN'), ('with', 'IN'), ('Australian', 'JJ'), ('trade', 'NN'), ('minister', 'NN'), ('Dan', 'NNP'), ('Tehan', 'NNP'), ('.', '.')]
```

```
(S
  (NP The/DT bilateral/JJ trade/NN)
  between/IN
  the/DT
  two/CD
  countries/NNS
  is/VBZ
  expected/VBN
  to/TO
```

```
rise/VB
from/IN
existing/VBG
27/CD
billion/CD
USD/NNP
to/TO
45/CD
billion/CD
USD/NNP
in/IN
the/DT
next/JJ
five/CD
years/NNS
,/,
(NP union/NN)
(NP minister/NN)
Piyush/NNP
Goyal/NNP
said/VBD
as/IN
he/PRP
signed/VBD
(NP the/DT deal/NN)
with/IN
(NP Australian/JJ trade/NN)
(NP minister/NN)
Dan/NNP
Tehan/NNP
./.)
```

```
(NP The/DT bilateral/JJ trade/NN)
(NP union/NN)
(NP minister/NN)
(NP the/DT deal/NN)
```

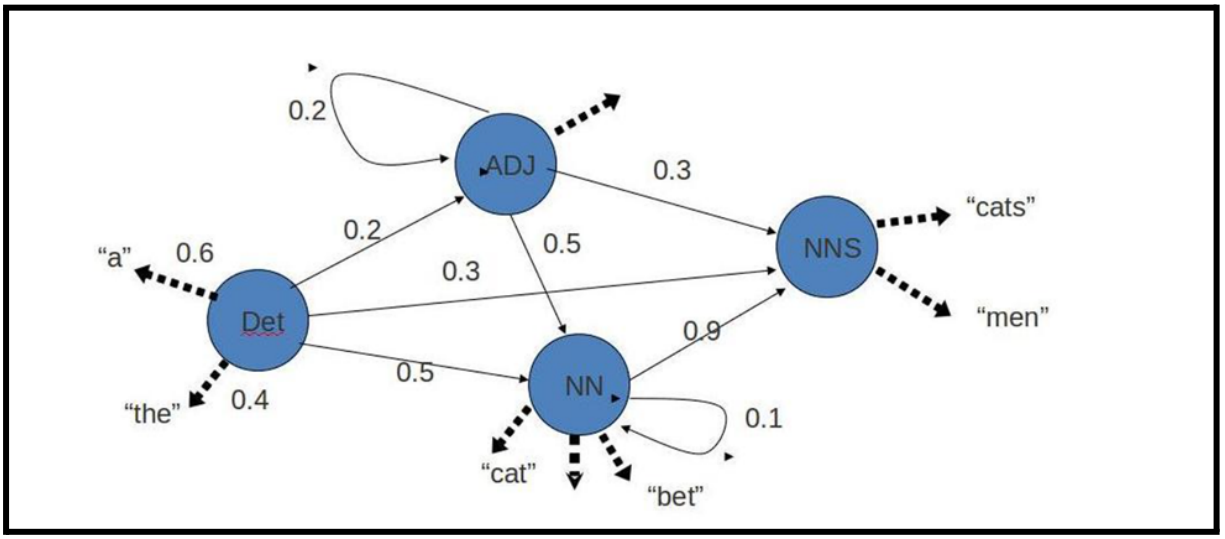
```
(NP Australian/JJ trade/NN)
(NP minister/NN)
[('two', 'CARDINAL'),
 ('27 billion USD to', 'MONEY'),
 ('45 billion USD', 'MONEY'),
 ('the next five years', 'DATE'),
 ('Piyush Goyal', 'PERSON'),
 ('Australian', 'NORP'),
 ('Dan Tehan', 'PERSON')]
```

Conclusion:
Thus, we have successfully performed chunking, POS tagging and Named Entity Recognition.

EXPERIMENT - 4

Aim: To study and implement Hidden Markov Models (HMM) to calculate the probability of a sequence of tags using NLTK

Theory:
A Hidden Markov Model (HMM) is a statistical Markov model in which the system being modelled is assumed to be a Markov process with unobserved (hidden) states. In a regular Markov model, the state is directly visible to the observer, and therefore the state transition probabilities are the only parameters. In a hidden Markov model, the state is not directly visible, but output, dependent on the state, is visible.



Hidden Markov Model has two important components:

1. Transition Probabilities: The one-step transition probability is the probability of transitioning from one state to another in a single step.
2. Emission Probabilities: The output probabilities for an observation from state. Emission probabilities $B = \{b_{i,k} = b_i(o_k) = P(o_k|q_i)\}$, where o_k is an Observation. Informally, B is the probability that the output is given that the current state is q_i

For POS tagging; it is assumed that POS are generated as random processes, and each process randomly generates a word. Hence, transition matrix denotes the

transition probability from one POS to another and emission matrix denotes the probability that a given word can have a particular POS.

Code:

```
import nltk
from nltk.corpus import brown
nltk.download('brown')
```

```
brown_word_tags = []
for brown_sent in brown.tagged_sents():
  brown_word_tags.append(('SOS' ,'START'))

  for word, tag in brown_sent:
    brown_word_tags.append((tag[:2], word))

  brown_word_tags.append(('EOS','END'))

brown_word_tags[:5]
```

```
[('SOS', 'START'),
 ('AT', 'The'),
 ('NP', 'Fulton'),
 ('NN', 'County'),
 ('JJ', 'Grand')]
```

```
cfd_tag_words = nltk.ConditionalFreqDist(brown_word_tags)
cpd_tag_words = nltk.ConditionalProbDist(cfd_tag_words, nltk.LaplaceProbDist)
```

```
print (f" The probability of an adjective (JJ) being 'smart' is {cpd_tag_words['JJ'].prob('smart'):.6f}")
```

```
The probability of an adjective (JJ) being 'smart' is 0.000260
```

```
print (f" The probability of an verb (VB) being 'try' is {cpd_tag_words['VB'].prob('try'):.6f}")
```

```
The probability of an verb (VB) being 'try' is 0.000991
```

```
brown_tags = [tag for tag, words in brown_word_tags]
cfd_tags = nltk.ConditionalFreqDist(nltk.bigrams(brown_tags))
cpd_tags = nltk.ConditionalProbDist(cfd_tags, nltk.LaplaceProbDist)
print (f" The probability of DT occuring after NN is {cpd_tags['NN'].prob('DT'):.6f}")
print (f" The probability of VB occuring after NN is {cpd_tags['NN'].prob('VB'):.6f}")
```

```
The probability of DT occuring after NN is 0.001839

The probability of VB occuring after NN is 0.064627
```

```
prob_tag_sequence = cpd_tags['SOS'].prob('PP') * cpd_tag_words['PP'].prob('She') * \
cpd_tags['PP'].prob('VB') * cpd_tag_words['VB'].prob('loves') * \
cpd_tags['VB'].prob('JJ') * cpd_tag_words['JJ'].prob('spicy') * \
cpd_tags['JJ'].prob('NN') * cpd_tag_words['NN'].prob('food') * \
cpd_tags['NN'].prob('EOS')
```

```
print("The probability of sentence 'She loves spicy food' having the tag sequence 'START PP VB JJ NN END' is : ", prob_tag_sequence)
```

```
The probability of sentence 'She loves spicy food' having the tag sequence 'START PP VB JJ NN END' is :  9.601527367873185e-20
```

```
prob_tag_sequence = cpd_tags['SOS'].prob('PP') * cpd_tag_words['PP'].prob('I') * \
cpd_tags['PP'].prob('VB') * cpd_tag_words['VB'].prob('want') * \
cpd_tags['VB'].prob('TO') * cpd_tag_words['TO'].prob('to') * \
cpd_tags['TO'].prob('VB') * cpd_tag_words['VB'].prob('race') * \
cpd_tags['VB'].prob('EOS')

print("The probability of sentence 'I want to race' having the tag sequence 'START PP VB TO VB END' is : ", prob_tag_sequence)
```

```
The probability of sentence 'I want to race' having the tag sequence 'START PP VB TO VB END' is :  1.1313534426303036e-14
```

Conclusion:

Thus,we have studied the Hidden Markov Model and computed the probability tag sequence using HMMs

EXPERIMENT - 5

Aim: Perform Morphological Analysis on a word.

Theory:

Morphemes are considered as smallest meaningful units of language. These morphemes can either be a root word(play) or affix(-ed). Combination of these morphemes is called morphological process. So, word "played" is made out of 2 morphemes "play" and "-ed".

Thus finding all parts of a word(morphemes) and thus describing properties of a word is called "Morphological Analysis". For example, "played" has information verb "play" and "past tense", so given word is past tense form of verb "play".

A morphological analyzer is a program that analyzes the morphology of an input word. It uses rules to identify the root and grammatical features of given words. It splits a given word into it's root, lexical category, gender, number, person, case, case marker or tense aspect modality(TAM), suffix and other required grammatical features as given below.

1. root : Root of the word (e.g. ladZake word has root ladZakA)

2. cat : Category of the word (e.g. Noun=n, Pronoun=pn, Adjective=adj, verb=v, adverb=adv

post-position=psp and avvya=avy)

3. gen : Gender of the word

4. num : Number of the word (e.g. Singular=sg, Plural=pl, dual, and

any)

5. per : Person of the word (e.g. 1st Person=1, 2nd Person=2, 3rd Person=3, and any)

6. case : Case of the word (e.g. direct=d, oblique=o and any)

7. tam : Case marker for noun or Tense Aspect Mood(TAM) for verb of the word

8. suff : Suffix of the word

Analysis of a word

बच्चों (bachchoM) = बच्चा (bachchaa) (root) + ओं (oM) (suffix)

(ओं=3 plural oblique)

A linguistic paradigm is the complete set of variants of a given lexeme. These variants can be classified according to shared inflectional categories (eg: number, case etc) and arranged into tables.

Paradigm for बच्चा

case/ num	singular	plural
direct	बच्चा(bachchaa)	बच्चे(bachche)
oblique	बच्चे(bachche)	बच्चों (bachchoM)

Algorithm to get बच्चों **(bachchoM) from** बच्चा **(bachchaa)**

1. Take Root बच्च (bachch) आ (aa)

2. Delete आ (aa)

3. output बच्च (bachch)

4. Add ओं (oM) to output

5. Return बच्चों (bachchoM)

Therefore, आ is deleted and ओं is added to get बच्चों

Add-Delete table for बच्चा

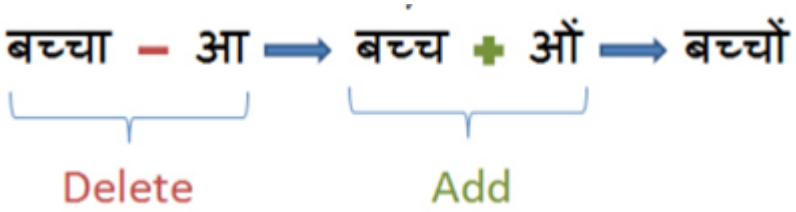
Delete	Add	Number	Case	Variants
आ(aa)	आ(aa)	sing	dr	बच्चा(bachchaa)
आ(aa)	ए(e)	plu	dr	बच्चे(bachche)
आ(aa)	ए(e)	sing	ob	बच्चे(bachche)
आ(aa)	ओं(oM)	plu	ob	बच्चों(bachchoM)

Paradigm Class

Words in the same paradigm class behave similarly, for Example लड़क is in the same paradigm class as बच्च, so लड़का would behave similarly as बच्चा as they share the same paradigm class.

बच्चा	-ओं
लड़का	-ओं
play	-ed
want	-ed

Words can be analyzed morphologically if we know all variants of a given root word. We can use an 'Add-Delete' table for this analysis.



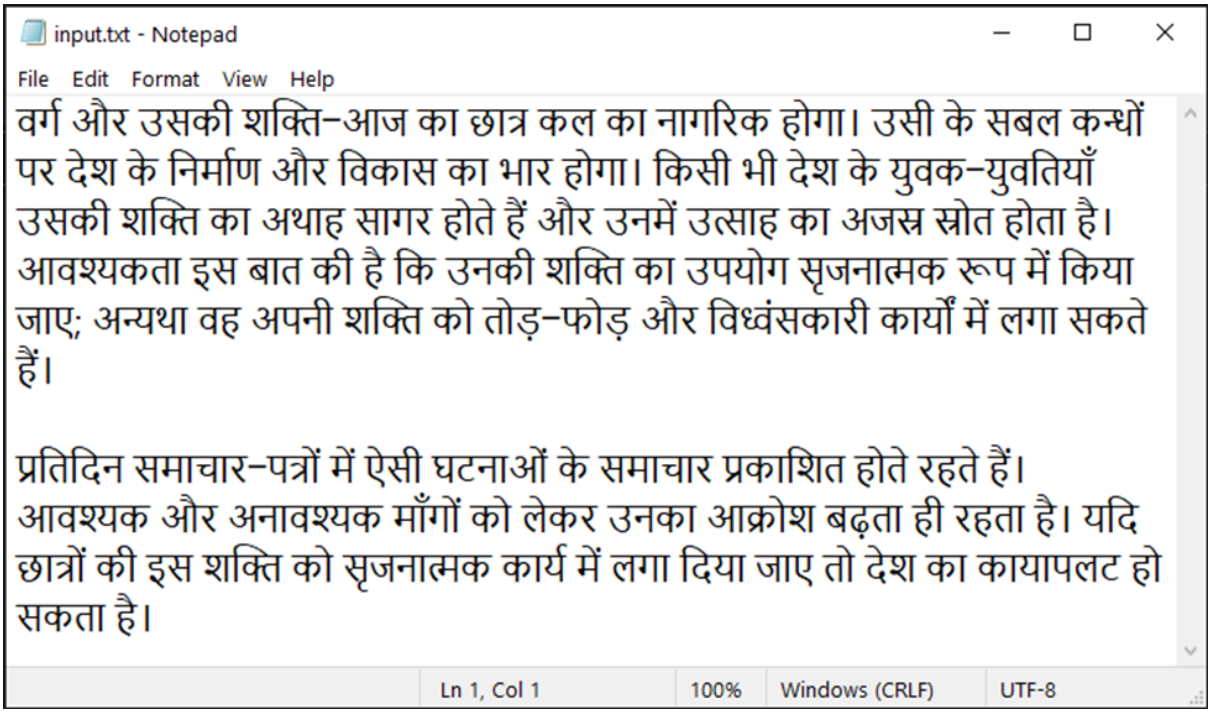
Code:

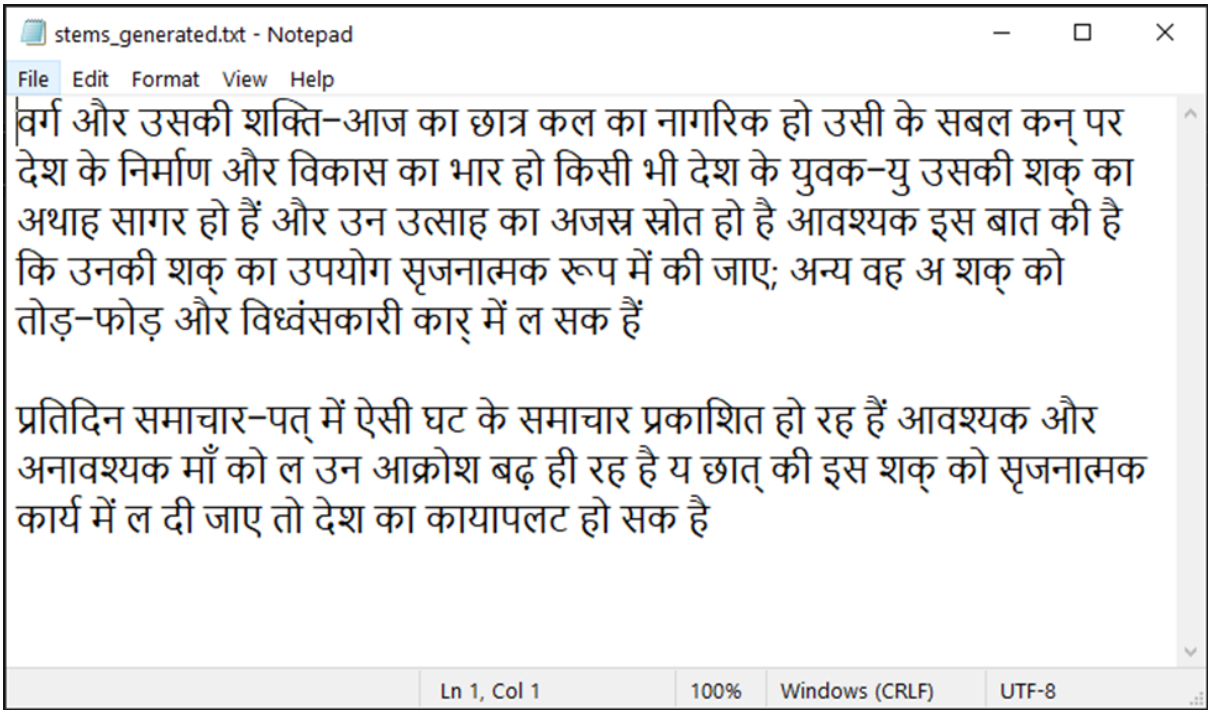
```
import codecs
import re

# read the input file
filepath = "input.txt"
f = codecs.open(filepath, encoding="utf-8")
text = f.read()

sentences = text.split(u"।")  # since hindi sentences end with '।'
words_list = list()
for sentence in sentences:
        words = sentence.split(" ")  # words are seperated by a space in hindi
        words_list += words

suffixes = {
        1: [
        u"ाएगी",
        u"ाएगा",
        u"ाओगी",
        u"ाओगे",
        u"एंगी",
        u"ेंगी",
        u"एंगे",
        u"ेंगे",
```

```
        u"ूंगी",
        u"ूंगा",
        u"ातीं",
        u"नाओं",
        u"नाएं",
        u"ताओं",
        u"ताएं",
        u"ियाँ",
        u"ियों",
        u"ियां",
        ],
        2: [u"ो", u"े", u"ू", u"ु", u"ीय", u"ि", u"ा"],
        3: [u"कर", u"ाओ", u"िए", u"ाई", u"ाए", u"ने", u"नी", u"ना", u"ते", u"ीं", u"ती", u"ता", u"ाँ", u"ां", u"ों", u"ें"],
        4: [
        u"ाकर",
        u"ाइए",
        u"ाईं",
        u"ाया",
        u"ेगी",
        u"ेगा",
        u"ोगी",
        u"ोगे",
        u"ाने",
        u"ाना",
        u"ाते",
        u"ाती",
        u"ाता",
        u"तीं",
        u"ाओं",
        u"ाएं",
        u"ुओं",
        u"ुएं",
        u"ुआं",
        ],
        5: [u"ाएंगी", u"ाएंगे", u"ाऊंगी", u"ाऊंगा", u"ाइयाँ", u"ाइयों", u"ाइयां"],
}
```

```
stems = list()
for word in words_list:
        for L in range(1, 5):
        if len(word) >= L + 1:
        for suffix in suffixes[L]:
        if word.endswith(suffix):
                word = word[:-L]  # stripping the suffix from the word
                try:
                if word[-1] == u"ि":
                  word = word[:-1] + u"ी"
                except:
                print(word)
        if word:
        stems.append(word)

filename = "stems_generated.txt"
f = codecs.open(filename, "w", encoding="utf-8")  # open in write mode
for stem in stems:
        f.write(str(stem))
        f.write(u"\u0020")
f.close()
```

Output:





Conclusion: We have successfully performed morphological analysis on a Hindi text.

EXPERIMENT - 6

Aim:

1. Find synonyms and hyponyms using Python nltk and WordNet
2. Get all the hyponyms and hypernyms for a given word
3. Get all Hyponyms with synsetID
4. Implement Word sense disambiguation using Sysnet

Theory:

Wordnet is a large collection of words and vocabulary from the English language that are related to each other and are grouped in some way. That's the reason WordNet is also called a lexical database. WordNet groups nouns, adjectives, verbs which are similar and calls them synsets or synonyms. A group of synsets might belong to some other synset. Synset is a special kind of a simple interface that is present in NLTK to look up words in WordNet. Synset instances are the groupings of synonymous words that express the same concept. For example, the synsets "Brick" and "concrete" belong to the synset "Construction Materials" or the synset "Brick" also belongs to another synset called "brickwork ". In the example given, brick and concrete are called hyponyms of synset construction materials and also the synsets construction material and brickwork are called synonyms.

Word Sense Disambiguation is an important method of NLP by which the meaning of a word is determined, which is used in a particular context. NLP systems often face the challenge of properly identifying words, and determining the specific usage of a word in a particular sentence has many applications.
Word Sense Disambiguation basically solves the ambiguity that arises in determining the meaning of the same word used in different situations.

How to implement WSD?

There are four main ways to implement WSD.

- **Dictionary- and knowledge-based methods**

- These methods rely on text data like dictionaries, thesaurus, etc. It is based on the fact that words that are related to each other can be found in the definitions. The popularly used Lesk method, which is a seminal dictionary-based method.

- **Supervised methods**

  - In this type, sense-annotated corpora are used to train machine learning models. But a problem that may arise is that such corpora are very tough and time-consuming to create.

- **Semi-supervised Methods**

  - Due to the lack of such corpus, most word sense disambiguation algorithms use semi-supervised methods. The process starts with a small amount of data, which is often manually created.

  - This is used to train an initial classifier. This classifier is used on an untagged part of the corpus, to create a larger training set. Basically, this method involves bootstrapping from the initial data, which is referred to as the seed data.

  - Semi-supervised methods thus, use both labeled and unlabelled data.

- **Unsupervised Methods**

  - Unsupervised Methods pose the greatest challenge to researchers and NLP professionals. A key assumption of these models is that similar meanings and senses occur in a similar context. They are not dependent on manual efforts, hence can overcome the knowledge acquisition deadlock.

Lesk Algorithm

Lesk Algorithm is a classical Word Sense Disambiguation algorithm introduced by Michael E. Lesk in 1986.

The Lesk algorithm is based on the idea that words in a given region of the text will have a similar meaning. In the Simplified Lesk Algorithm, the correct meaning of each word context is found by getting the sense which overlaps the most among the given context and its dictionary meaning.

Code:

```
import nltk
from nltk.corpus import wordnet as wn

nltk.download("omw-1.4")
nltk.download("wordnet")

synonyms = []
word = "call"
for syn in wn.synsets(word):
        for i in syn.lemmas():
        synonyms.append(i.name())
print(f"Synonyms of {word} are: ")
print(set(synonyms), "\n")

hyponyms = []
print(f"Hyponyms of {word} are: ")
for syn in wn.synsets(word):
        for i in syn.hyponyms():
    hyponyms.append(i.name().split(".")[0])
print(set(hyponyms), "\n")

hypernyms = []
for syn in wn.synsets(word):
        # print(syn.hypernym_distances())
        for i in syn.hypernym_distances():
        hypernyms.append((i[0].name().split(".")[0], i[1]))
print(f"Hypernyms of {word} are: ")
print(set(hypernyms), "\n")
```

```
vehicle = wn.synset("call.n.01")
typesOfVehicles = list(set([w for s in vehicle.closure(lambda s: s.hyponyms()) for w in s.lemma_names()]))
print(f"Hyponyms of Synset ID {vehicle} are: ")
print(set(typesOfVehicles))
```

```
import nltk

nltk.download("omw-1.4")
nltk.download("wordnet")
from nltk.corpus import wordnet as wn
from nltk.wsd import lesk

sentence = ["I", "went", "to", "the", "track", "to", "watch", "the", "car", "race", "."]

print(lesk(sentence, "track", "n"), "\n")
print(lesk(sentence, "track"), "\n")

syns = wn.synsets("track")
print(syns, "\n")

# print the word
print(syns[0].lemmas()[0].name(), "\n")

from nltk.corpus import wordnet as wn

for ss in wn.synsets("track"):
    print(ss, ss.definition())

print("\n")

print([(s, s.pos()) for s in wn.synsets("can")], "\n")

sentence = "I can kill seventy monsters in a minute".split()
print(lesk(sentence, "can"), "\n")
```

```
print(lesk(sentence, "can", pos="v"), "\n")

print(lesk("Ram adores Site".split(), "adores", synsets=[]))
```

Output:

```
Synonyms of call are:
{'cry', 'margin_call', 'scream', 'telephone', 'bid', 'prognosticate', 'send_for', 'call_option', 'phone', 'name', 'song', 'yell', 'ring', 'telephone_call', 'visit', 'anticipate', 'birdcall', 'call_off', 'phone_call', 'hollo', 'squall', 'address', 'holler', 'vociferation', 'claim', 'Call', 'call_up', 'call', 'shout_out', 'foretell', 'shout', 'outcry', 'promise', 'call_in', 'predict', 'birdsong', 'forebode'}

Hyponyms of call are:
{'howl', 'crank_call', 'second-guess', 'call-back', 'read', 'scream', 'beep', 'conference_call', 'term', 'bell-like_call', 'hoot', 'entitle', 'summon', 'yelling', 'preempt', 'yodel', 'style', 'baptize', 'prophesy', 'double', 'forecast', 'clamor', 'long_distance', 'shriek', 'catcall', 'hollo', 'war_cry', 'misname', 'underbid', 'hail', 'call-in', 'rename', 'raise', 'lift', 'local_call', 'function_call', 'cell_phone', 'outbid', 'call_up', 'tag', 'boo', 'squawk', 'hurrah', 'overbid', 'wake-up_call', 'halloo', 'dub', 'muster', 'augur', 'round', 'outcall', 'system_call', 'two-note_call', 'collect_call', 'blue_murder', 'recall', 'call_in', 'bellow', 'hosanna', 'whoop', 'refer', 'drop_by', 'noise', 'bet', 'see', 'post'}

Hypernyms of call are:
{('shout', 0), ('message', 3), ('attitude', 2), ('writing', 5), ('communicate', 4), ('order', 1), ('communication', 3), ('transfer', 4), ('visit', 1), ('call_option', 0), ('act', 3), ('call', 0), ('ask', 3), ('break', 3), ('interact', 4), ('entice', 1), ('transfer', 6), ('request', 1), ('address', 0), ('act', 5), ('medium', 3), ('instrumentality', 4), ('assembly', 3), ('utter', 1), ('think', 2), ('postpone', 1), ('communicate', 8), ('inform', 2), ('entity', 5), ('communication', 7), ('abstraction', 4), ('request', 3), ('name', 0), ('coding_system', 3), ('convey', 9), ('psychological_feature', 7), ('think', 4), ('move', 11), ('interact', 8), ('event', 4), ('transfer', 10), ('abstraction', 6), ('evaluate', 2), ('legal_document', 3), ('declare', 1), ('request', 5), ('talk', 2), ('option', 1), ('writing', 4), ('challenge', 1), ('artifact', 5), ('communicate', 3), ('code', 2), ('event', 6), ('decision', 1), ('speech_act', 2),
```

('abstraction', 8), ('designate', 2), ('request', 7), ('derivative_instrument', 2), ('convey', 4), ('challenge', 3), ('meeting', 2), ('move', 6), ('invite', 4), ('ask', 2), ('communicate', 5), ('written_communication', 6), ('demand', 2), ('transfer', 5), ('birdcall', 0), ('predict', 0), ('talk', 6), ('provoke', 2), ('convey', 6), ('indicate', 1), ('group_action', 4), ('psychological_feature', 4), ('move', 8), ('communicate', 7), ('wait', 3), ('request', 2), ('whole', 6), ('margin_call', 0), ('entity', 7), ('cry', 0), ('physical_entity', 8), ('speculate', 2), ('signal', 2), ('change', 5), ('psychological_feature', 6), ('think', 3), ('stop', 1), ('ask', 6), ('act', 9), ('choice', 2), ('cognition', 3), ('request', 4), ('label', 1), ('entity', 9), ('visit', 0), ('end', 4), ('oppose', 3), ('document', 4), ('reason', 3), ('telephone', 1), ('event', 5), ('interact', 3), ('evaluate', 3), ('communication', 4), ('telecommunication', 2), ('object', 7), ('convey', 3), ('see', 1), ('inclination', 1), ('move', 5), ('ask', 1), ('act', 4), ('written_communication', 5), ('demand', 1), ('telecommunicate', 1), ('instruction', 1), ('contest', 2), ('communication', 6), ('transfer', 7), ('address', 1), ('convey', 5), ('change', 2), ('move', 7), ('delay', 2), ('meet', 1), ('animal_communication', 1), ('entity', 6), ('claim', 2), ('awaken', 1), ('abstraction', 5), ('play', 1), ('auditory_communication', 2), ('argue', 4), ('converse', 5), ('bid', 0), ('psychological_feature', 5), ('utterance', 1), ('interrupt', 2), ('communicate', 2), ('read', 1), ('entity', 8), ('compete', 2), ('guess', 1), ('abstraction', 7), ('action', 3)}

Hyponyms of Synset ID Synset('call.n.01') are:
{'call-in', 'crank_call', 'call-back', 'trunk_call', 'conference_call', 'collect_call', 'local_call', 'toll_call', 'long_distance', 'three-way_calling', 'wake-up_call', 'long-distance_call'}

[nltk_data] Downloading package omw-1.4 to /root/nltk_data...
[nltk_data] Downloading package wordnet to /root/nltk_data...
Synset('track.n.11')

Synset('track.n.11')

[Synset('path.n.04'), Synset('lead.n.03'), Synset('track.n.03'), Synset('racetrack.n.01'), Synset('cut.n.08'), Synset('track.n.06'), Synset('track.n.07'), Synset('track.n.08'), Synset('track.n.09'), Synset('track.n.10'), Synset('track.n.11'), Synset('track.v.01'), Synset('track.v.02'), Synset('chase.v.01'), Synset('traverse.v.01'), Synset('track.v.05')]

path

Synset('path.n.04') a line or route along which something travels or moves
Synset('lead.n.03') evidence pointing to a possible solution
Synset('track.n.03') a pair of parallel rails providing a runway for wheels
Synset('racetrack.n.01') a course over which races are run
Synset('cut.n.08') a distinct selection of music from a recording or a compact disc
Synset('track.n.06') an endless metal belt on which tracked vehicles move over the ground
Synset('track.n.07') (computer science) one of the circular magnetic paths on a magnetic disk that serve as a guide for writing and reading data
Synset('track.n.08') a groove on a phonograph recording
Synset('track.n.09') a bar or pair of parallel bars of rolled steel making the railway along which railroad cars or other vehicles can roll
Synset('track.n.10') any road or path affording passage especially a rough one
Synset('track.n.11') the act of participating in an athletic competition involving running on a track
Synset('track.v.01') carry on the feet and deposit
Synset('track.v.02') observe or plot the moving path of something
Synset('chase.v.01') go after with the intent to catch
Synset('traverse.v.01') travel across or pass over
Synset('track.v.05') make tracks upon

[(Synset('can.n.01'), 'n'), (Synset('can.n.02'), 'n'), (Synset('can.n.03'), 'n'), (Synset('buttocks.n.01'), 'n'), (Synset('toilet.n.02'), 'n'), (Synset('toilet.n.01'), 'n'), (Synset('can.v.01'), 'v'), (Synset('displace.v.03'), 'v')]

Synset('can.v.01')

Synset('can.v.01')

None

Conclusion:Hence, word sense disambiguation has been performed using WordNet, sysnet and NLTK.

EXPERIMENT - 7

Aim: Plagiarism Detection Using NLP

Theory:

Natural Language Processing technologies can be used to effectively to detect plagiarism in texts. NLP distance measures can be applied to detect external plagiarism, i.e., when both the original text as well as the suspicious text are available.

Steps

1. Before using other NLP techniques, we first apply pre-processing techniques to the text. change all the uppercase alphabets to lowercase to generalize tokens across both the texts. Further, Stop-Words like 'or', 'the' and 'in' and punctuations are removed, as these are functional in nature and do not give any extra information about the document.
2. Next, we read the original and the suspicious (possibly plagiarized) documents.
3. The plagiarism content between the two texts is found by calculating the Jaccard similarity coefficient.
4. Another method is finding the Longest Common Subsequence (LCS) in the texts.
5. Evaluate all the scores on the documents in the dataset. There are three types of documents: near copy, lightly revised and heavily revised.

Code:

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

```
from nltk.corpus import stopwords
from nltk.corpus import wordnet as wn

doc1 = "When your focus is to improve employee performance, it's essential to
encourage ongoing dialogue between managers and their direct reports. Some
companies encourage supervisors to hold one-on-one meetings with employees as
a way to facilitate two-way communication."
doc2 = "When your focus is to improve employee performance, ongoing dialogue
between managers and their direct reports is essential. While performance
management often involves conducting annual performance evaluations, it does
involve more than just that."

from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize

stop_words = set(stopwords.words("english"))
word_tokens1 = word_tokenize(doc1)
word_tokens2 = word_tokenize(doc2)

filtered_sentence1 = [w for w in word_tokens1 if not w.lower() in stop_words]
filtered_sentence2 = [w for w in word_tokens2 if not w.lower() in stop_words]

filtered_sentence1 = []
for w in word_tokens1:
    if w not in stop_words:
        filtered_sentence1.append(w)
filtered_sentence2 = []
for w in word_tokens2:
    if w not in stop_words:
        filtered_sentence2.append(w)

print(word_tokens1)
print(filtered_sentence1)
print(word_tokens2)
print(filtered_sentence2)
s1 = " ".join(filtered_sentence1)
s2 = " ".join(filtered_sentence2)
```

```
from nltk.tokenize import RegexpTokenizer

tokenizer = RegexpTokenizer(r"\w+")
s1 = tokenizer.tokenize(s1)
s2 = tokenizer.tokenize(s2)
s1 = " ".join(s1)
s2 = " ".join(s2)
jd_sent_1_2 = nltk.jaccard_distance(set(s1), set(s2))
print(f"Similarity using Jaccard Similarity {(1 - jd_sent_1_2)*100}%")


def lcs(l1, l2):
    s1 = word_tokenize(l1)
    s2 = word_tokenize(l2)
    # storing the dp values
    dp = [[None] * (len(s1) + 1) for i in range(len(s2) + 1)]
    for i in range(len(s2) + 1):
        for j in range(len(s1) + 1):
            if i == 0 or j == 0:
                dp[i][j] = 0
            elif s2[i - 1] == s1[j - 1]:
                dp[i][j] = dp[i - 1][j - 1] + 1
            else:
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])
    return dp[len(s2)][len(s1)]


from nltk.tokenize import sent_tokenize, word_tokenize

tokens_o = word_tokenize(doc1)
tokens_p = word_tokenize(doc2)
sent_o = sent_tokenize(doc1)
sent_p = sent_tokenize(doc2)

# maximum length of LCS for a sentence in suspicious text
max_lcs = 0
```

```
sum_lcs = 0
for i in sent_p:
    for j in sent_o:
        l = lcs(i, j)
        max_lcs = max(max_lcs, l)
    sum_lcs += max_lcs
    smax_lcs = 0

score = sum_lcs / len(tokens_p)
print(f"Similarity using LCS {score*100}%")
```

Output:

```
['When', 'your', 'focus', 'is', 'to', 'improve', 'employee', 'performance', ',', 'it', '"s",
'essential', 'to', 'encourage', 'ongoing', 'dialogue', 'between', 'managers', 'and', 'their',
'direct', 'reports', '.', 'Some', 'companies', 'encourage', 'supervisors', 'to', 'hold',
'one-on-one', 'meetings', 'with', 'employees', 'as', 'a', 'way', 'to', 'facilitate', 'two-way',
'communication', '.']
['When', 'focus', 'improve', 'employee', 'performance', ',', '"s", 'essential', 'encourage',
'ongoing', 'dialogue', 'managers', 'direct', 'reports', '.', 'Some', 'companies', 'encourage',
'supervisors', 'hold', 'one-on-one', 'meetings', 'employees', 'way', 'facilitate', 'two-way',
'communication', '.']
['When', 'your', 'focus', 'is', 'to', 'improve', 'employee', 'performance', ',', 'ongoing',
'dialogue', 'between', 'managers', 'and', 'their', 'direct', 'reports', 'is', 'essential', '.',
'While', 'performance', 'management', 'often', 'involves', 'conducting', 'annual',
'performance', 'evaluations', ',', 'it', 'does', 'involve', 'more', 'than', 'just', 'that', '.']
['When', 'focus', 'improve', 'employee', 'performance', ',', 'ongoing', 'dialogue',
'managers', 'direct', 'reports', 'essential', '.', 'While', 'performance', 'management',
'often', 'involves', 'conducting', 'annual', 'performance', 'evaluations', ',', 'involve', '.']
Similarity using Jaccard Similarity 91.30434782608697%
Similarity using LCS 94.73684210526315%
```

Conclusion: Hence, plagiarism checker has been performed using Jaccard Similarity and LCS.

EXPERIMENT - 8

Aim: Sentence generation using context-free grammar

Theory:

Context Free Grammar

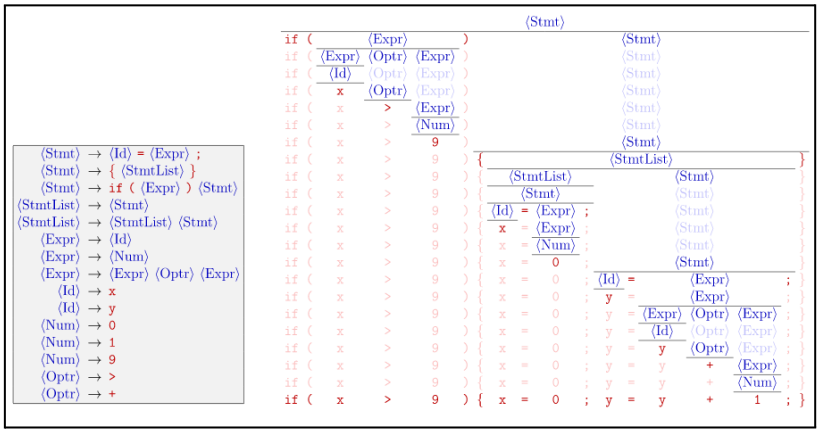
Context-free grammars (CFGs) are used to describe context-free languages. A context-free grammar is a set of recursive rules used to generate patterns of strings. A context-free grammar can describe all regular languages and more, but they cannot describe all possible languages.

Context-free grammar (CFG) is a formal grammar whose production rules can be applied to a nonterminal symbol regardless of its context. In a context-free grammar, each production rule is of the form. $A \to \alpha$

with A a single nonterminal symbol, and α a string of terminals and/or nonterminal (α can be empty). A formal grammar is essentially a set of production rules that describe all possible strings in a given formal language. Production rules are simple replacements. For example, the first rule in the picture,

⟨Stmt⟩ → ⟨Id⟩ = ⟨Expr⟩

replaces ⟨Stmt⟩ with ⟨Id⟩ = ⟨Expr⟩ There can be multiple replacement rules for a given nonterminal symbol. The language generated by a grammar is the set of all strings of terminal symbols that can be derived, by repeated rule applications, from some particular nonterminal symbol ("start symbol"). Nonterminal symbols are used during the derivation process, but do not appear in its result string.



Code:

```
from collections import defaultdict
import random

class CFG(object):
    def __init__(self):
        self.rules = defaultdict(list)

    def add_production_rule(self, lhs, rhs):
        productions = rhs.split('|')
        for rules in productions:
            self.rules[lhs].append(tuple(rules.split()))

    def generate_random(self, symbol):
        sentence = ''

        random_production = random.choice(self.rules[symbol])

        for sym in random_production:
            if sym in self.rules:
                sentence += self.generate_random(sym)
            else:
                sentence += sym + ' '

        return sentence

cfg = CFG()
cfg.add_production_rule('S', 'NP VP')
cfg.add_production_rule('NP', 'Det N')
cfg.add_production_rule('PP', 'P NP')
cfg.add_production_rule('VP', 'V NP | V PP | VP')
cfg.add_production_rule('Det', 'the | a')
cfg.add_production_rule('N', 'man | elephant | dog | cat')
cfg.add_production_rule('P', 'in | with | on')
cfg.add_production_rule('V', 'slept | saw | walked | kicked | followed | shot')
```

```
print(dict(cfg.rules))


random_sentences = []
for i in range(10):
   sentence = cfg.generate_random('S')
   if sentence not in random_sentences:
      random_sentences.append(sentence)
random_sentences
```

Output:

```
{'S': [('NP', 'VP')], 'NP': [('Det', 'N')], 'PP': [('P', 'NP')], 'VP': [('V', 'NP'), ('V', 'PP'), ('VP',)],
'Det': [('the',), ('a',)], 'N': [('man',), ('elephant',), ('dog',), ('cat',)], 'P': [('in',), ('with',), ('on',)],
'V': [('slept',), ('saw',), ('walked',), ('kicked',), ('followed',), ('shot',)]}


['the elephant walked in a cat ',
 'the man walked with a cat ',
 'a man kicked a man ',
 'a dog kicked with a man ',
 'a elephant saw in a dog ',
 'a dog saw with a cat ',
 'a man walked with a elephant ',
 'the elephant kicked in the elephant ',
 'the elephant saw in the dog ',
 'the elephant shot in a dog ']
```

Conclusion:

Context-free grammar provides a set of rules that define the generation of a sentence structure in a language. These sets of rules generate patterns of strings that satisfy the language rules. The rules are defined in terms of set of variables that are recursively defined in terms of another, the final rules thus do not require knowledge of context or symbols that may or not be present. Thus, these symbols can be substituted in terms of another leading to the CFG production rules.