# Experiment No.1

Name:-Yash S. Oswal Roll no.:-61

## **Title:** Pre-processing: Sentence and Word Tokenization

**Aim:** To write a program to implement Sentence and Word Tokenization

**Theory:**

**Tokenization** is the process of tokenizing or splitting a string, text into a list of tokens. Tokens are parts like a word is a token in a sentence, and a sentence is a token in a paragraph.

Before processing a natural language, we need to identify the *words* that constitute a string of characters. That’s why tokenization is the most basic step to proceed with NLP (textdata). This is important because the meaning of the text could easily be interpreted by analyzing the words present in the text.

NLTK contains a module called tokenize() which further classifies into two sub-categories:

* **Word tokenize:** We use the word\_tokenize() method to split a sentence into tokens or words
* **Sentence tokenize:** We use the sent\_tokenize() method to split a document or paragraph into sentences

**Conclusion:** NLTK provides us with modules for word and sentence tokenization. Tokenization is first basic step in Natural Language Processing**.**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Program**  **Execution (8M)** | **Documentation (1M)** | **Timely**  **Submission (2M)** | **Viva (4M)** | **Total (15M)** | **Sign** |
|  |  |  |  |  |  |

Experiment 1

Sentence and Word Tokenization

### Code: Word Tokenization

import nltk

filename="marathi.txt" data=""

with open(filename,"r",encoding="utf-8-sig") as fd: data = fd.read(1001)

word\_tokens=nltk.word\_tokenize(data) print(word\_tokens)

### Output:

**Code: Sentence Tokenization**

import nltk

filename="marathi.txt" data=""

with open(filename,"r",encoding="utf-8-sig") as fd: data = fd.read(50000)

sent\_tokens=nltk.sent\_tokenize(data) print(sent\_tokens)

### Output:

**Experiment No.2**

**Title:** Pre-processing: Filtration, Script Validation, Stop Word Removal in python

**Aim:** To write a program to implement Filtration, Script Validation, Stop Word Removal

**Theory:**

Stopwords are the most common words in any natural language. For the purpose of analyzing text data and building NLP models, these stopwords might not add much value to the meaning of the document.

**Removing stopwords is not a hard and fast rule in NLP. It depends upon the task that we are working on.** For tasks like text classification, where the text is to be classified into different categories, stopwords are removed or excluded from the given text so that more focus can be given to those words which define the meaning of the text.

## On removing stopwords, dataset size decreases and the time to train the model also decreases

* Removing stopwords can potentially help improve the performance as there are fewer and only meaningful tokens left. Thus, it could increase classification accuracy
* Even search engines like Google remove stopwords for fast and relevant retrieval of data from the database

We can remove stopwords while performing the following tasks:

* Text Classification
  + Spam Filtering
  + Language Classification
  + Genre Classification
* Caption Generation
* Auto-Tag Generation

Avoid removing stopwords during:

* Machine Translation
* Language Modeling
* Text Summarization
* Question-Answering problems

**Conclusion:** Words such as articles and some verbs are usually considered **stop words** because they don't help us to find the context or the true meaning of a sentence. After tokenization, stopwords removal can be performed. NLTK provides us with list of stopwords stored in 16 different languages.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Program**  **Execution (8M)** | **Documentation (1M)** | **Timely**  **Submission (2M)** | **Viva (4M)** | **Total (15M)** | **Sign** |
|  |  |  |  |  |  |

Experiment 2

Filtration, Script Validation, Stop Word Removal

### Code: Stopword removal for English words

from nltk.corpus import stopwords

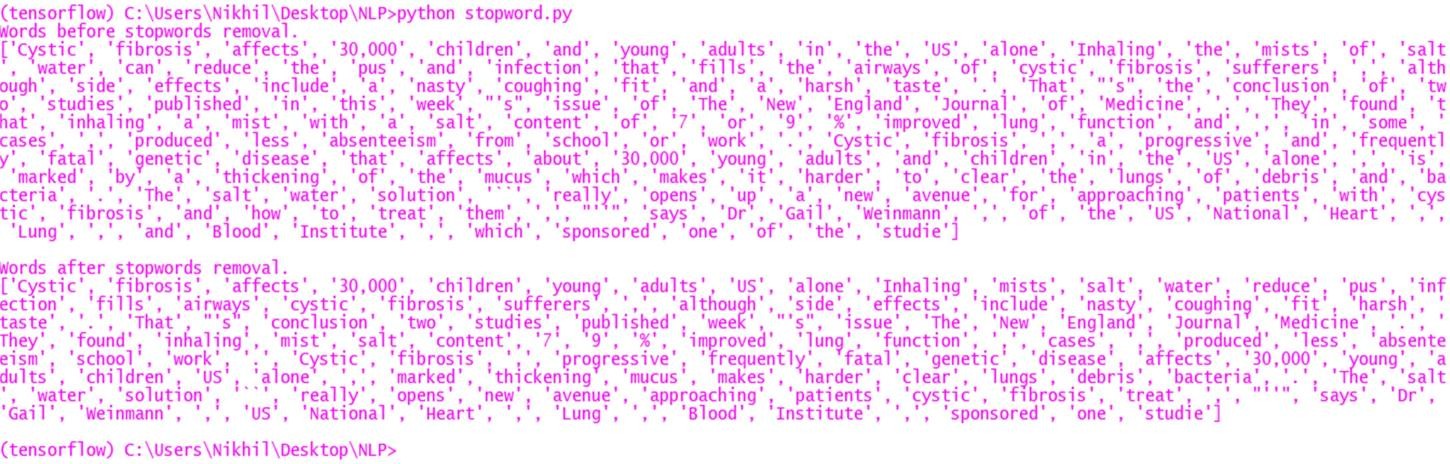
from nltk.tokenize import word\_tokenize stop\_words = set(stopwords.words('english'))

with open("C:/Users/sakec.307-004/AppData/Roaming/nltk\_data/corpora/abc/science.txt") as fd: data = fd.read(2000)

word\_tokens = word\_tokenize(data) print(word\_tokens)

filtered\_sentence = [w for w in word\_tokens if w not in stop\_words] print(filtered\_sentence)

### Output:



**Code: Stopword removal for Marathi words**

from nltk.corpus import stopwords

from nltk.tokenize import word\_tokenize

mr\_stopwords = []

with open("C:/Users/sakec.307-004/Desktop/mr\_stopwords.txt",encoding="utf-8-sig") as fd: mr\_stopwords = fd.read().split("\n")

mr\_stopwords = mr\_stopwords print("Marathi stopwords\n") print(mr\_stopwords,"\n") word\_tokens = []

with open("C:/Users/sakec.307-004/Desktop/marathi.txt",encoding="utf-8-sig") as fd: data = fd.read()

print("Marathi Tokens\n") word\_tokens = word\_tokenize(data) print(word\_tokens,"\n")

filtered\_sentence = [w for w in word\_tokens if w not in mr\_stopwords] print("MArathi tokenes without stopwords\n\n") print(filtered\_sentence)

word\_tokens = filtered\_sentence punctuation = [",",".","?","!","\"","\'",":",";"]

filtered\_sentence = [w for w in word\_tokens if w not in punctuation]

print("\nMarathi tokens without punctuation\n") print(filtered\_sentence)

### Output:



**Experiment No.3**

**Title:** Stemming & Lemmatization

## **Aim:** To write a program to implement stemming, lemmatization.

**Theory:**

Stemming is a kind of normalization for words. Normalization is a technique where a set of words in a sentence are converted into a sequence to shorten its lookup. The words which have the same meaning but have some variation according to the context or sentence are normalized.

In another word, there is one root word, but there are many variations of the same words. For example, the root word is "eat" and it's variations are "eats, eating, eaten and like so".

Lemmatization is the algorithmic process of finding the lemma of a word depending on their meaning. Lemmatization usually refers to the morphological analysis of words, which aims to remove inflectional endings. It helps in returning the base or dictionary form of a word, which is known as the lemma. The NLTK Lemmatization method is based on WorldNet's built-in morph function. Text preprocessing includes both stemming as well as lemmatization.

Stemming algorithm works by cutting the suffix from the word. In a broader sense cuts either the beginning or end of the word.

On the contrary, Lemmatization is a more powerful operation, and it takes into consideration morphological analysis of the words. It returns the lemma which is the base form of all its inflectional forms. In-depth linguistic knowledge is required to create dictionaries and look for the proper form of the word. Stemming is a general operation while lemmatization is an intelligent operation where the proper form will be looked in the dictionary. Hence, lemmatization helps in forming better machine learning features.

**Conclusion:** Stemming and Lemmatization both generate the root form of the inflected words. The difference is that stem might not be an actual word whereas, lemma is an actual language word

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Program**  **Execution (8M)** | **Documentation (1M)** | **Timely**  **Submission (2M)** | **Viva (4M)** | **Total (15M)** | **Sign** |
|  |  |  |  |  |  |

Experiment 3 Stemming and Lemmatization

### Code: Stemming & Lemmatization

## English Language

from nltk.tokenize import word\_tokenize  
from nltk.stem import PorterStemmer, WordNetLemmatizer  
  
stemmer = PorterStemmer()  
lemmatizer = WordNetLemmatizer()  
  
sentence = '''This is a sentence for stemming and lemmatization.   
He went out for jogging while listening to musical instruments'''  
  
tokens = word\_tokenize(sentence.lower())  
  
print("Sentence: ", sentence)  
print("Tokens: ", tokens)  
  
stems = [stemmer.stem(word) for word in tokens]  
lemmas = [lemmatizer.lemmatize(word) for word in tokens]  
  
print("Stemmed Sentence: ", \*stems)  
print("Stemmed Tokens: ", stems)  
  
print("Lemmatized Sentence: ", \*lemmas)  
print("Lemmatized Tokens: ", lemmas)

## Regional Language

1. Stemming:

import nltk as nl

import codecs

def generate\_stem\_words(word):

suffixes = {1:[u"ो",u"े",u"ू",u"ु",u"ी",u"ि",u"ा"],2:[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"ें"],3: [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"],4: [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"],5: [u"ाएंगी",u"ाएंगे",u"ाऊंगी",u"ाऊंगा",u"ाइयाँ",u"ाइयों",u"ाइयां"],}

for L in 5, 4, 3, 2, 1:

if len(word) > L + 1:

for suf in suffixes[L]:

#print type(suf),type(word),word,suf

if word.endswith(suf):

return word[:-L]

return word

io = "mnm.txt"

op = ""

with codecs.open(io,encoding='utf-8-sig',mode='r') as i:

op=i.read()

print("Sentence: \n\n "+op)

stemmed = []

for i in nl.word\_tokenize(op):

stemmed.append(generate\_stem\_words(i))

print("After stemming: \n\n")

print(stemmed)

1. Lemmatization:

import nltk as nl

import codecs

inp = "final.txt"

op = ""

with codecs.open(inp,encoding='utf-8-sig',mode='r') as i:

op=i.read()

print("Sentence : \n"+op)

lemma\_list = "lemma2.txt"

lemmas = ""

with codecs.open(lemma\_list,encoding='utf-8-sig',mode='r') as i:

lemmas=i.read()

lemma\_dict={}

lemmatized = ""

for line in lemmas.splitlines():

line=line.split(":")

lemma\_dict[line[1]]=line[0]

for word in nl.word\_tokenize(op):

le=word.strip()

for key, value in lemma\_dict.items():

if(value.strip() == word.strip()):

le=key.strip()

lemmatized += " "+le

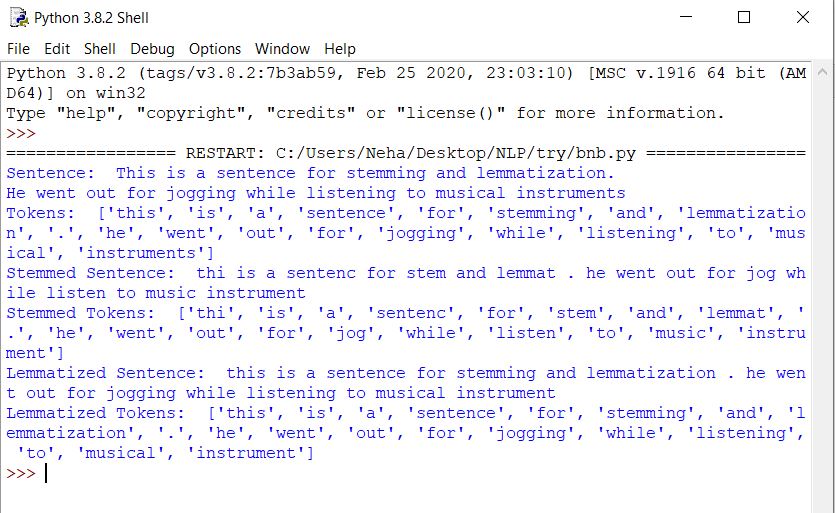
print("\n After lemmatization:")

for line in lemmatized.split("|"):

print(line+"|")

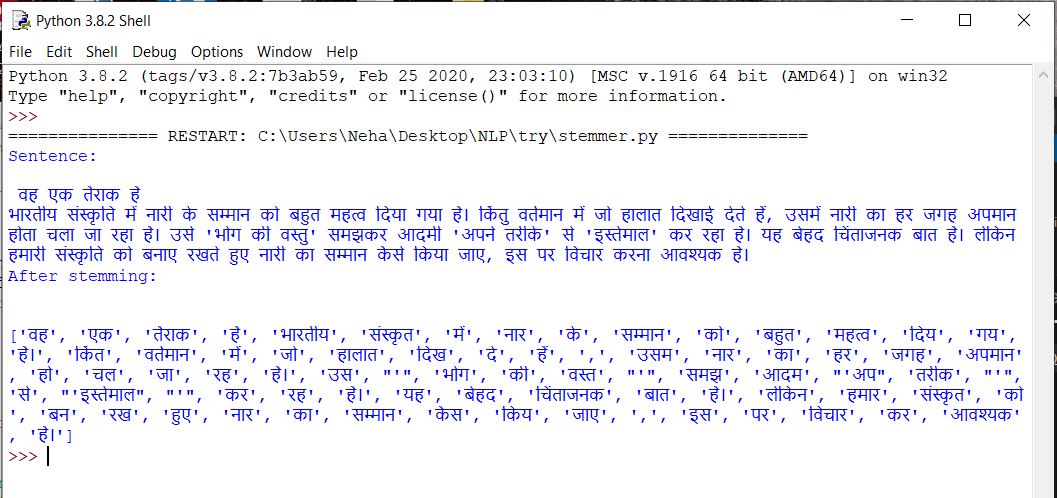
**Output:**

## English Language

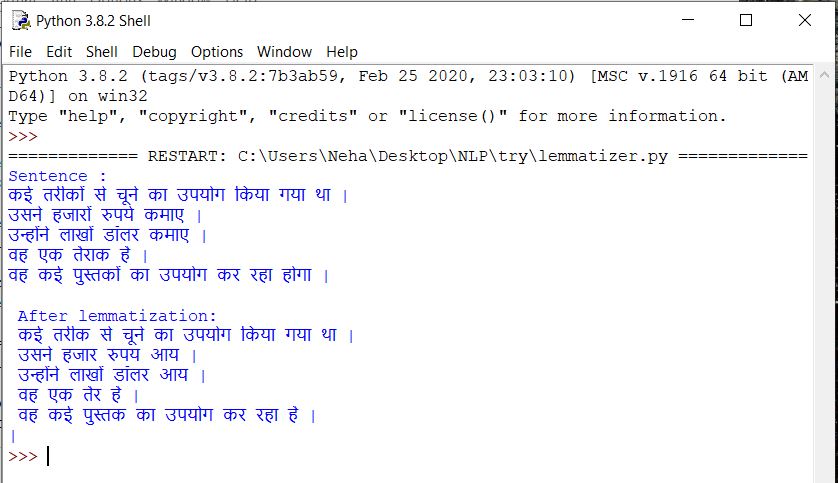
****

1. Regional Language

## Stemming

****

## Lemmatization

****

**Experiment No.4**

**Title:** Morphological Analysis

## **Aim:** To write a program for implement morphological analysis

**Theory:**

Morphology is the study of the way words are built up from smaller meaning bearing units i.e., morphemes. A morpheme is the smallest meaningful linguistic unit.

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".

**Conclusion:** Morphology studies how words can be structured and formed. Morphemes are primitive unit of meaning in a language. Morphological analysis helps us with understanding existing words as well as forming new ones.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Program Execution (8M)** | **Documentation (1M)** | **Timely Submission (2M)** | **Viva (4M)** | **Total (15M)** | **Sign** |
|  |  |  |  |  |  |

Experiment 4 Morphology

### Code:

1. English Language

from nltk import pos\_tag, word\_tokenize, WordNetLemmatizer  
  
lemmatizer = WordNetLemmatizer()  
  
sentence = "They refuse to permit us to obtain the refuse permit"  
words = word\_tokenize(sentence)  
lemmas = [lemmatizer.lemmatize(word) for word in words]  
  
print("Lemmas: ", lemmas)  
print("POS tags: ", pos\_tag(words))

1. Regional Language

import nltk

from nltk.tag import tnt

from nltk.corpus import indian

train\_data = indian.tagged\_sents('hindi.pos')

tnt\_pos\_tagger = tnt.TnT()

tnt\_pos\_tagger.train(train\_data)

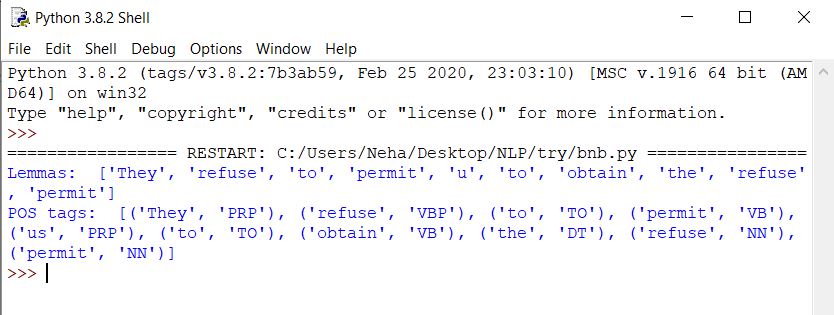
text = "इराक के विदेश मंत्री ने अमरीका के उस प्रस्ताव का मजाक उड़ाया है , जिसमें अमरीका ने संयुक्त राष्ट्र के प्रतिबंधों को इराकी नागरिकों के लिए कम हानिकारक बनाने के लिए कहा है ।"

tagged\_words = (tnt\_pos\_tagger.tag(nltk.word\_tokenize(text)))

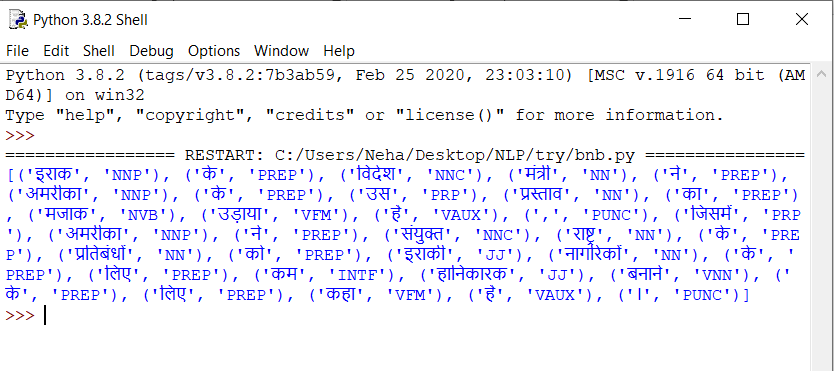
print(tagged\_words)

### Output:

1. English Language



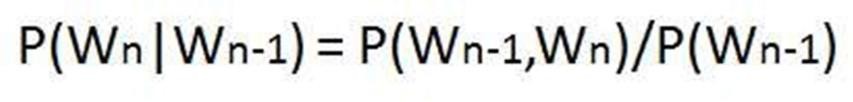
1. Regional Language



**Title:** N-gram model

# Experiment No.5

## **Aim:** Write a program to implement N-gram model in python to predict sentence probability



**Theory:**

A combination of words forms a sentence. However, such a formation is meaningful only when the words are arranged in some order.

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

Here, Wn refers to the word token corresponding to the nth word in a sequence.

**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.

**Conclusion:** N-grams of texts are extensively used in text mining and natural language processing tasks. N-gram model can be used for auto completion of sentences or auto spell check by calculating probabilities of words or senteces.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Program Execution (8M)** | **Documentation (1M)** | **Timely Submission (2M)** | **Viva (4M)** | **Total (15M)** | **Sign** |
|  |  |  |  |  |  |

Experiment 5 N-gram

### Code:

import nltk

from nltk import word\_tokenize from nltk.corpus import stopwords from nltk.util import ngrams

from collections import Counter #import seaborn as sns

import pandas as pd

punctuation = [",",".","?","!","\"","\'",":",";","\'\'","\"\""]

fd = open("data.txt","r") text = fd.read()

word = word\_tokenize(text)

word = [w for w in word if w not in stopwords.words('english')] word = [w for w in word if w not in punctuation] word\_set=set(word)

unigram = [(w,word.count(w)) for w in word\_set] print("unigram",unigram)

bigram\_list = [(word[i],word[i+1]) for i in range(len(word)-2)] bigram\_set = set(bigram\_list)

bigram = [(w,bigram\_list.count(w)) for w in bigram\_set] print("bigram",bigram)

matrix=[]

for i in word\_set: temp=[]

for j in word\_set: temp.append(bigram\_list.count((i,j))/word.count(i))

matrix.append(temp) print(matrix)

df = pd.DataFrame(matrix) df.columns=word\_set.copy() df.index=word\_set.copy()

print(df)

test\_str = input("Enter the test string :") word\_test = word\_tokenize(test\_str)

bigram\_list\_test = [(word[i],word[i+1]) for i in range(len(word)-2)] p\_value = 1

for i in bigram\_list\_test: p\_value\*=bigram\_list.count(i)/word.count(i[0])

print("Probability value of sentence :",p\_value)

### Output:

**Experiment No. 6**

**Title:** POS tagging using HMM

## **Aim:** Write a program to implement POS tagging using HMM in python

**Theory:**

A Hidden Markov Model (HMM) is a statistical Markov model in which the system being modeled 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 Probabilties: : The output probabilities for an observation from state. Emission probabilities B = { bi,k = bi(ok) = P(ok | qi) }, where okis an Observation. Informally, B is the probability that the output is ok given that the current state is qi

For POS tagging, it is assumed that POS are generated as random process, 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. Word acts as the observations. Some of the basic assumptions are:

1. First-order (bigram) Markov assumptions:
   1. Limited Horizon: Tag depends only on previous tag P(ti+1 = tk | t1=tj1,ti=tji) = P(ti+1 = tk | ti = tj)
   2. Time invariance: No change over time

P(ti+1 = tk | ti = tj) = P(t2 = tk | t1 = tj) = P(tj -> tk)

2. Output probabilities:

Probability of getting word wk for tag tj: P(wk | tj) is independent of other tags or words

**Conclusion:** Part-of-Speech tagging is an important part of many natural language processing pipelines where the words in a sentence are marked with their respective parts of speech. HMM model treats input tokens to be observable sequence while tags are considered as hidden states and goal is to determine the hidden state sequence.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Program**  **Execution (8M)** | **Documentation (1M)** | **Timely**  **Submission (2M)** | **Viva (4M)** | **Total (15M)** | **Sign** |
|  |  |  |  |  |  |

### Code:

import nltk

from nltk import word\_tokenize from nltk.corpus import stopwords from nltk.util import ngrams

from collections import Counter import pandas as pd

from nltk.stem import WordNetLemmatizer from nltk.corpus import wordnet

Experiment 6

POS tagging using HMM

fd = open("data2.txt",'r') text = fd.read()

word\_list = word\_tokenize(text) text = text.lower()

word\_set = list(set(word\_tokenize(text))) word\_set.sort()

updated\_word\_list = list(filter(lambda a:a!='EOS', word\_list)) pos = nltk.pos\_tag(updated\_word\_list)

text = text.lower()

word\_list = word\_tokenize(text) word\_set = list(set(word\_list)) word\_set.sort()

pos\_updated=[] j=0

for i in word\_list: if i=='eos':

pos\_updated.append(('eos','Eos')) else:

pos\_updated.append(pos[j]) j+=1

print("Word with tag\n",pos\_updated)

word\_pos = []

for w in pos\_updated: if(w[1] in ['JJ','JJR','JJS']):

word\_pos.append((w[0].lower(), "Adjective")) elif(w[1] in ['VB','VBD','VBG','VBN','VBP','VBZ']):

word\_pos.append((w[0].lower(), "Verb"))

elif(w[1] in ['NN','NNS','NNP','NNPS']):

word\_pos.append((w[0].lower(), "Noun"))

elif(w[1] in ['RB','RBR','RBS']):

word\_pos.append((w[0].lower(), "Adverb")) else:

word\_pos.append((w[0].lower(),w[1])) print(word\_pos)

pos\_list = []

for i in word\_pos: pos\_list.append(i[1])

print("\nTag list\n",pos\_list)

pos\_bigram = []

for i in range(len(pos\_list)-1): pos\_bigram.append((pos\_list[i], pos\_list[i+1]))

print("\nTag bigram\n",pos\_bigram)

word\_pos\_set = list(set(word\_pos)) word\_pos\_set.sort()

pos\_set = list(set(pos\_list))

pos\_set.sort()

pos\_bigram\_set = list(set(pos\_bigram)) pos\_bigram\_set.sort()

word\_pos\_dict=dict() for i in word\_pos\_set:

word\_pos\_dict[i] = word\_pos.count(i) print("\nWord with tag dictionary\n",word\_pos\_dict)

pos\_dict=dict() for i in pos\_set:

pos\_dict[i]=pos.count(i) print("\nTag dictionary\n",pos\_dict)

pos\_bigram\_dict = dict() for i in pos\_bigram\_set:

pos\_bigram\_dict[i] = pos\_bigram.count(i)

print("\nPos bigram dictionary\n",pos\_bigram\_dict,"\n\n")

emission\_matrix = [] for i in pos\_set:

temp=[]

for j in word\_set: temp.append(word\_pos.count((j,i))/word\_list.count(j))

emission\_matrix.append(temp) print("Emission Matrix\n")

df = pd.DataFrame(emission\_matrix) df.columns=word\_set.copy() df.index=pos\_set.copy() print(df,"\n\n")

transition\_matrix = [] for i in pos\_set:

temp=[]

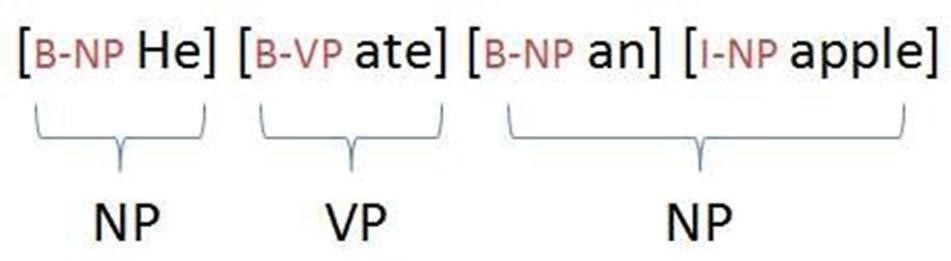
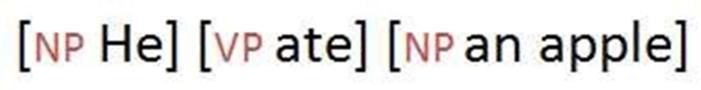
for j in pos\_set: temp.append(pos\_bigram.count((j,i))/pos\_list.count(j))

transition\_matrix.append(temp) print("Transition Matrix\n")

df = pd.DataFrame(transition\_matrix) df.columns=pos\_set.copy() df.index=pos\_set.copy()

print(df)

### Output:



**Experiment No. 7**

**Title:** Chunking

## **Aim:** Write a program to implement Rule based Chunking in python

**Theory:**

Chunking is a process of extracting phrases from unstructured text.

Chunking of text invloves dividing a text into syntactically correlated words. For example, the sentence 'He ate an apple.' can be divided as follows:

Each chunk has an open boundary and close boundary that delimit the word groups as a minimal non-recursive unit. This can be formally expressed by using IOB prefixes.

**Conclusion:** Chunking is a process of extracting phrases from unstructured text, which means analyzing a sentence to identify the constituents (Noun Groups, Verbs, verb groups, etc.)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Program Execution (8M)** | **Documentation (1M)** | **Timely Submission (2M)** | **Viva (4M)** | **Total (15M)** | **Sign** |
|  |  |  |  |  |  |

Experiment 7 Chunking

### Code:

## English Language

from nltk.chunk.regexp import ChunkString, ChunkRule, ChinkRule  
from nltk.tree import Tree  
  
tree = Tree('S', [('the', 'DT'), ('book', 'NN'),  
 ('has', 'VBZ'), ('many', 'JJ'), ('chapters', 'NNS')])  
  
print("Initial Tree: ")  
tree.pretty\_print()  
chunk\_string = ChunkString(tree)  
  
chunk\_rule = ChunkRule('<DT><NN.\*><.\*>\*<NN.\*>', 'chunk determiners and nouns')  
chunk\_rule.apply(chunk\_string)  
  
ir = ChinkRule('<VB.\*>', 'chink verbs')  
ir.apply(chunk\_string)  
tree = chunk\_string.to\_chunkstruct()  
  
print("After chunking: ")  
tree.pretty\_print()

## Regional Language

## import nltk

from nltk.corpus import Indian

from nltk.tag import tnt

import string

from nltk import RegexpParser

tagged\_set = 'hindi.pos'

## word\_set = indian.sents(tagged\_set)

## count = 0

## for sen in word\_set:

## count = count + 1

## sen = "".join([" "+i if not i.startswith("'") and i not in string.punctuation else i for i in sen]).strip()

## train\_perc = .9

## train\_rows = int(train\_perc\*count)

## test\_rows = train\_rows + 1

## data = indian.tagged\_sents(tagged\_set)

## train\_data = data[:train\_rows]

## test\_data = data[test\_rows:]

## pos\_tagger = tnt.TnT()

## pos\_tagger.train(train\_data)

## pos\_tagger.evaluate(test\_data)

## text = "एक दिन प्रात:काल की बात है कि दो बालक राह किनारे खड़े तर्क कर रहे थे।"

## tokenized = nltk.word\_tokenize(text)

## tag = pos\_tagger.tag(tokenized)

## print(tag)

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

## cp = nltk.RegexpParser(grammar)

## result = cp.parse(tag)

## print(result)

## result.draw()

### Output:

## English Language

# 

## Regional Language

# 

# 

**Experiment No. 8**

**Title:** Named Entity Recognition

## **Aim:** Write a program to implement Named Entity Recognition for given corpus in python

**Theory:**

Named Entity Recognition is a process where an algorithm takes a string of text (sentence or paragraph) as input and identifies relevant nouns (people, places, and organizations) that are mentioned in that string.

**Named-entity recognition** (**NER**) (also known as **entity identification**, **entity**

## **chunking** and **entity extraction**) is a sub-task of information extraction that seeks to locate and classify named entities in text into pre-defined categories such as the names of persons, organizations, locations, expressions of times, quantities, monetary values, percentages, etc.

NER systems have been created that use linguistic grammar-based techniques as well as statistical models such as machine learning. Hand-crafted grammar-based systems typically obtain better precision, but at the cost of lower recall and months of work by experienced computational linguists. Statistical NER systems typically require a large amount of manually annotated training data. Semi-supervised approaches have been suggested to avoid part of the annotation effort.

(NER)is probably the first step towards information extraction that seeks to locate and classify named entities in text into pre-defined categories such as the names of persons,

organizations, locations, expressions of times, quantities, monetary values, percentages, etc. NER is used in many fields in Natural Language Processing (NLP)

Major Application areas of NER:

Classifying content for news providers

News and publishing houses generate large amounts of online content on a daily basis and managing them correctly is very important to get the most use of each article. Named Entity

Recognition can automatically scan entire articles and reveal which are the major people, organizations, and places discussed in them. Knowing the relevant tags for each article help in automatically categorizing the articles in defined hierarchies and enable smooth content discovery.

Efficient Search Algorithms

Let’s suppose you are designing an internal search algorithm for an online publisher that has millions of articles. If for every search query the algorithm ends up searching all the words in millions of articles, the process will take a lot of time. Instead, if Named Entity Recognition can be run once on all the articles and the relevant entities (tags) associated with each of those articles are stored separately, this could speed up the search process considerably. With this approach, a search term will be matched with only the small list of entities discussed in each article leading to faster search execution.

Customer Support

There are a number of ways to make the process of customer feedback handling smooth and Named Entity Recognition could be one of them.

**Conclusion**: Named entity recognition (NER) is an entity chunking/extraction technique, which is popularly used in information extraction. Common entity tags include PERSON, LOCATION and ORGANIZATION. NER is useful for extracting relevant information from unstructured information.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Program Execution (8M)** | **Documentation (1M)** | **Timely Submission (2M)** | **Viva (4M)** | **Total (15M)** | **Sign** |
|  |  |  |  |  |  |

Experiment 8 Named Entity Recognition

### Code:

1. English Language

from nltk import pos\_tag, ne\_chunk

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

train\_text = "Shooting is a very popular sport. Hunters use guns to shoot animals. Terrorists also use them to kill"

sample\_text = "Sharpshooter Mark shoots a dangerous animal with a gun"

tokenized = sent\_tokenize(sample\_text)

def process\_content():

for i in tokenized[:5]:

words = word\_tokenize(i)

tagged = pos\_tag(words)

namedEnt = ne\_chunk(tagged)

namedEnt.draw()

process\_content()

1. Regional Language

text = “ ”

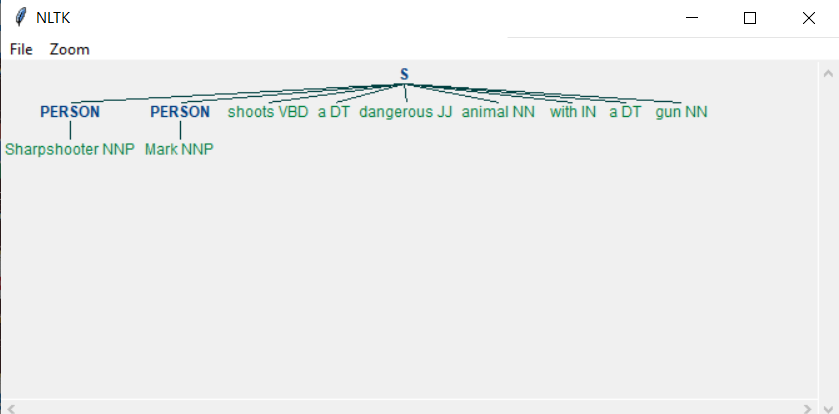
from polygot.text import Text

ip\_hindi\_text = Text(text)

print(ip\_hindi\_text.entities)

### Output:

1. English Language



1. Regional Language

