VIT | Vidyalankar Institute of Technology

Accredited A+ by NAAC

VIDYALANKAR INSTITUTE OF TECHNOLOGY

DEPARTMENT OF COMPUTER ENGINEERING

Lab Manual

Subject: NATURAL LANGUAGE PROCESSING

SEM-VIII

Prof. Mohini Chaudhari

2020-2021

Subject	Natural Language Processing
Semester	VIII
Academic Year	2020-21
Software Requirements	Python, NLTK
Hardware Requirements	Desktops
Theory Faculty In-charge	Prof. Mohini Chaudhari
Practical Faculty In-charge	Prof. Mohini Chaudhari
Laboratory	314
Teaching Assistant	--
Revised On	18 January, 2021
Prepared By	Prof. Mohini Chaudhari
Sign	
Endorsed By HOD	Dr. Sachin Bojewar

Vidyalankar Institute of Technology

DEPARTMENT OF COMPUTER ENGINEERING

LAB CODE

1. Students should report to the concerned labs as per the timetable schedule.
2. Students who turn up late to the labs will in no case be permitted to perform the experiment scheduled for the day.
3. After completion of the experiment, certification of the concerned staff in-charge in the observation book is necessary.
4. Students should bring a notebook of about 100 pages and should enter the readings/observations into the notebook while performing the experiment.
5. The record of observations along with the detailed experimental procedure of the experiment performed in the immediate last session should be submitted and certified by the faculty member.
6. The group-wise division made in the beginning should be adhered to, and no mix up of student among different groups will be permitted later.
7. The components required pertaining to the experiment should be collected from the concerned Lab Assistants.
8. When the experiment is completed, students should disconnect the setup made by them, and should return all the components/instruments taken for the purpose. Any damage of the equipment or burn-out of components will be viewed seriously either by putting penalty or by dismissing the total group of students from the lab for the semester/year.

Students should be present in the labs for the total scheduled duration. Students are required to prepare thoroughly to perform the experiment coming to Laboratory. Procedure sheets/data sheets provided to the student's groups should be maintained neatly and to be returned after the experiment.

Experiment List

Practical No.	Title of the **Regular Experiments**	Concepts to be highlighted	CO Map
1	To perform text pre-processing	Word Analysis	CO2
2	To generate word forms from root and suffix information – Morphological Analysis	Word Analysis	CO2
3	To learn and calculate bigrams from a given corpus and calculate probability of a sentence – N-Gram Model	N Gram Model	CO2
4	To implement tagging Parts of Speech using Penn Tree Bank.	POS Tagging	CO3
5	To calculate emission and transition matrix for tagging Parts of Speech using Hidden Markov Model.	POS Tagging	CO3
6	To understand the concept of chunking and get familiar with the basic chunk tagset.	Chunking	CO4

EXPERIMENT NO. 01

Aim: Write a program to perform Text Pre-processing using NLP Techniques.

Theory:

Text pre-processing is done to help to increase the accuracy of the NLP tasks.

Generally, there are 3 main components:

- Tokenization
- Normalization
- Noise removal

In a nutshell, 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. Normalization aims to put all text on a level playing field, e.g., converting all characters to lowercase. Noise removal cleans up the text, e.g., remove extra whitespaces.

Steps to be performed in pre-processing are as given below:

1. 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.

2. Remove extra whitespaces: 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. To do this, we use the module unidecode.

3. Convert accented characters to ASCII characters

4. 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.

5. Remove special characters

6. Lowercase all texts

7. Convert number words to numeric form: This step involves the conversion of number words to numeric form, e.g., seven to 7, to standardize text.

8. Remove numbers: Removing numbers may make sense for sentiment analysis since numbers contain no information about sentiments. However, if our NLP task is to extract the number of tickets ordered in a message to our chatbot, we will definitely not want to remove numbers.

9. Remove stop words: 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.

10. Lemmatization: Lemmatization is the process of converting a word to its base form, e.g., "caring" to "care".

Program:

```
from nltk.tokenize import sent_tokenize
from nltk.corpus import stopwords
from nltk import word_tokenize
import string

input_string = 'He asked, \"How are you?\" and went away. Then he never came back. 1234'

def script_validation(sentence: str) -> str:
        english = [chr(i) for i in range(ord('a'), ord('z') + 1)]
        english.append('.')
        english.append(' ')
        for character in sentence.lower():
                if character not in english and character not in string.punctuation:
```

```
                    sentence = sentence.replace(character, "")

def segment_sentence(sentence: str) -> list:
        return sent_tokenize(sentence)

def filter_sentence(sentences: list) -> list:
        filtered_sentence = []
        for sentence in sentences:
                filtered_sentence.append(sentence.translate(sentence.maketrans('','',
string.punctuation)))
        return filtered_sentence

def tokenize(sentences: list) -> list:
        filtered_words = []
        for sentence in sentences:
                words = word_tokenize(sentence)
                for word in words:
                        filtered_words.append(word)
        return filtered_words

def remove_stop_words(sentence: list) -> list:
        filtered_sentence = []
        stop_words = set(stopwords.words("english"))
        for word in sentence:
                if word not in stop_words:
                        filtered_sentence.append(word)
        return filtered_sentence

def preprocess(sentence: str) -> str:
        print ('\nSentence:', sentence)

        sentence.strip()
        print ('\nStripped Sentence:', sentence)

        sentence = script_validation(sentence)
        print ('\nScript Validated Sentence:', sentence)

        sentence = segment_sentence(sentence)
        print ('\nSegmented Sentence:', sentence)

        sentence = filter_sentence(sentence)
        print ('\nFiltered Sentence:', sentence)

        sentence = tokenize(sentence)
        print ('\nTokenized Sentence:', sentence)

        sentence = remove_stop_words(sentence)
        print ('\nSWR Sentence:', sentence)

preprocess(input_string)
```

Output:

Sentence: He asked, "How are you?" and went away. Then he never came back. 1234

Stripped Sentence: He asked, "How are you?" and went away. Then he never came back. 1234

Script Validated Sentence: He asked, "How are you?" and went away. Then he never came back.

Segmented Sentence: ['He asked, "How are you?"', 'and went away.', 'Then he never came back.']

Filtered Sentence: ['He asked How are you', 'and went away', 'Then he never came back']

Tokenized Sentence: ['He', 'asked', 'How', 'are', 'you', 'and', 'went', 'away', 'Then', 'he', 'never', 'came', 'back']

SWR Sentence: ['He', 'asked', 'How', 'went', 'away', 'Then', 'never', 'came', 'back']

Conclusion:

To preprocess your text simply means to bring your text into a form that is predictable and analyzable for your task.

One task's ideal preprocessing can become another task's worst nightmare. So, take note: text preprocessing is not directly transferable from task to task.

The Following Steps are the most important for pre-processing:

- Text Normalization
- Tokenization
- Remove Stop Words
- Stemming
- Lemmatization, etc.

EXPERIMENT NO. 02

Aim: To learn the morphological features of a word by analyzing it

Theory:

A word can be simple or complex. For example, the word 'cat' is simple because one cannot further decompose the word into smaller part. On the other hand, the word 'cats' is complex, because the word is made up of two parts: root 'cat' and plural suffix '-s'

Analysis of a word into root and affix(es) is called as Morphological analysis of a word. It is mandatory to identify root of a word for any natural language processing task. A root word can have various forms. For example, the word 'play' in English has the following forms: 'play', 'plays', 'played' and 'playing'. Hindi shows a greater number of forms for the word 'खेल' (khela) which is equivalent to 'play'. The forms of 'खेल'(khela) are the following:
खेल(khela),
खेला(khelaa),
खेली(khelii),
खेल ूंगा(kheluungaa),
खेल ूंगी(kheluungii),
खेलेगा(khelegaa),
खेलेगी(khelegii),
खेलते(khelate),
खेलती(khelatii),
खेलने(khelane),
खेलकर(khelakar)

Thus, we understand that the morphological richness of one language might vary from one language to another. Indian languages are generally morphologically rich languages and therefore morphological analysis of words becomes a very significant task for Indian languages.

Types of Morphology:
Morphology is of two types,

Inflectional morphology: Deals with word forms of a root, where there is no change in lexical category. For example, 'played' is an inflection of the root word 'play'. Here, both played' and 'play' are verbs.
Derivational morphology: Deals with word forms of a root, where there is a change in the lexical category. For example, the word form 'happiness' is a derivation of the word 'happy'. Here, 'happiness' is a derived noun form of the adjective 'happy'.

Morphological Features:
All words will have their lexical category attested during morphological analysis. A noun and pronoun can take suffixes of the following features: gender, number, person, case. For example, morphological analysis of a few words is given below:

Language	input: word	output: analysis
Hindi	लडके (ladake)	rt=लड़का(ladakaa), cat=n, gen=m, num=sg, case=obl
Hindi	लडके (ladake)	rt=लड़का(ladakaa), cat=n, gen=m, num=pl, case=dir
Hindi	लड़कों (ladakoM)	rt=लड़का(ladakaa), cat=n, gen=m, num=pl, case=obl
English	boy	rt=boy, cat=n, gen=m, num=sg
English	boys	rt=boy, cat=n, gen=m, num=pl

A verb can take suffixes of the following features: tense, aspect, modality, gender, number, person

Language	input:word	output:analysis
Hindi	हँसी(hansii)	rt=हँस(hans), cat=v, gen=fem, num=sg/pl, per=1/2/3 tense=past, aspect=pft
English	toys	rt=toy, cat=n, num=pl, per=3

'rt' stands for root. 'cat' stands for lexical category. The value of lexicat category can be noun, verb, adjective, pronoun, adverb, preposition. 'gen' stands for gender. The value of gender can be masculine or feminine. 'num' stands for number. The value of number can be singular (sg) or plural (pl). 'per' stands for person. The value of person can be 1, 2 or 3. The value of tense can be present, past or future. This feature is applicable for verbs. The value of aspect can be perfect (pft), continuous (cont) or habitual (hab). This feature is not applicable for verbs. 'case' can be direct or oblique. This feature is applicable for nouns. A case is an oblique case when a postposition occurs after noun. If no postposition can occur after noun, then the case is a direct case. This is applicable for Hindi but not English

as it doesn't have any postpositions. Some of the postpositions in Hindi are: का(kaa), की(kii), के(ke), क (ko), में(meM)

Program:

```
import re
import random
from nltk.corpus import names
import nltk
from nltk.tokenize import PunktSentenceTokenizer
from nltk.stem import PorterStemmer
from nltk.stem import LancasterStemmer

mystr = 'Robert wanted Lisa in his teams'
wordList = re.sub("[^\w]", " ",  mystr).split()
print("WordList: ",wordList)
posdict = {"PRP$":"Possessive Pronoun","NNP":"Proper Noun","VBD":"Verb","IN":"Preposition/Subordinating Conjunction","NNS":"Noun Plural"}

porter = PorterStemmer()
lancaster = LancasterStemmer()

print("\nPorter Stemmer")
for i in range(len(wordList)):
        print(porter.stem(wordList[i]))
print("\nLancaster Stemmer")
for i in range(len(wordList)):
        print(lancaster.stem(wordList[i]))

def namepos(pos):
  return posdict[pos]

def gender_features(word):
  return {'last_letter':word[-1]}

labeled_names = ([(name, 'M') for name in names.words('male.txt')]+[(name, 'F') for name in names.words('female.txt')])
random.shuffle(labeled_names)
featuresets = [(gender_features(n), gender) for (n, gender) in labeled_names]
train_set, test_set = featuresets[500:], featuresets[:500]
classifier = nltk.NaiveBayesClassifier.train(train_set)
pos = nltk.pos_tag(wordList)

for i in range(len(wordList)):
  print("\nWord:",wordList[i])
  print("Gender:",classifier.classify(gender_features(wordList[i])))
  print("POS:",namepos(pos[i][1]))
```

Output:

Word List: ['Robert', 'wanted', 'Lisa', 'in', 'his', 'teams']

Porter Stemmer
robert
want
lisa
in
hi
team

Lancaster Stemmer
robert
want
lis
in
his
team

Word: Robert
Gender: M
POS: Proper Noun

Word: wanted
Gender: M
POS: Verb

Word: Lisa
Gender: F
POS: Proper Noun

Word: in
Gender: M
POS: Preposition/Subordinating Conjunction

Word: his
Gender: M
POS: Possessive Pronoun

Word: teams
Gender: M
POS: Noun Plural

Conclusion:

- Many language processing applications need to extract the information encoded in the words
- Parsers which analyze sentence structure need to know/check agreement between subjects and verbs or Adjectives and nouns
- Information retrieval systems benefit from knowing what the stem of a word is
- Machine translation systems need to analyze words to their components and generate words with specific features in the target language

EXPERIMENT NO. 03

Aim: To study N Gram Model to calculate bigrams from a given corpus and calculate probability of a sentence.

Theory:

N-Grams

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.

E.g.: Corpus – (eos) You book a flight (eos) I read a book (eos) You read (eos)

	(eos)	you	book	a	flight	I	read
(eos)	0	0.5	0	0	0	0.25	0
you	0	0	0.5	0	0	0	0.5
book	0.5	0	0	0.5	0	0	0
a	0	0	0.5	0	0.5	0	0
flight	1	0	0	0	0	0	0
I	0	0	0	0	0	0	1
read	0.5	0	0	0.5	0	0	0

P((eos) you read a book (eos))

= P(you|eos) * P(read|you) * P(a|read) * P(book|a) * P(eos|book)

= 0.5 * 0.5 * 0.5 * 0.5 * 0.5

= 0.03125

Program:

```
import re

bigramProbability = []
uniqueWords = []

def preprocess(corpus):
    corpus = 'eos ' + corpus.lower()
    corpus = corpus.replace('.', ' eos')
    return corpus

def generate_tokens(corpus):
    corpus = re.sub(r'[^a-zA-Z0-9\s]', ' ', corpus)
    tokens = [token for token in corpus.split(" ") if token != ""]
    return tokens

def generate_word_counts(wordList):
    wordCount = {}
    for word in wordList:
        if word not in wordCount:
            wordCount.update({word: 1})
        else:
            wordCount[word] += 1
    return(wordCount)
```

```
def generate_ngrams(tokens):
	ngrams = zip(*[tokens[i:] for i in range(2)])
	return [" ".join(ngram) for ngram in ngrams]

def print_probability_table():
	print('\nBigram Probability Table:\n')
	for word in uniqueWords:
		print('\t', word, end = ' ')
	print()
	for i in range(len(uniqueWords)):
		print(uniqueWords[i], end = ' ')
		probabilities = bigramProbability[i]
		for probability in probabilities:
			print('\t', probability, end = ' ')
		print()

def generate_bigram_table(corpus):
	corpus = preprocess(corpus)
	tokens = generate_tokens(corpus)
	wordCount = generate_word_counts(tokens)
	uniqueWords.extend(list(wordCount.keys()))
	bigrams = generate_ngrams(tokens)
	print (bigrams)

	for firstWord in uniqueWords:
		probabilityList = []
		for secondWord in uniqueWords:
			bigram = firstWord + ' ' + secondWord
			probability = bigrams.count(bigram) / wordCount[firstWord]
			probabilityList.append(probability)
		bigramProbability.append(probabilityList)

	print_probability_table()

def get_probability(sentence):
	corpus = preprocess(sentence)
	tokens = generate_tokens(corpus)
	probability = 1
	for token in range(len(tokens) - 1):
		firstWord = tokens[token]
		secondWord = tokens[token + 1]
		pairProbability =
bigramProbability[uniqueWords.index(firstWord)][uniqueWords.index(secondWord)]
		print('Probability: {1} | {0} = {2}'.format(firstWord, secondWord, pairProbability))
		probability *= pairProbability
	print('Probability of Sentence:', probability)

corpus = 'You book a flight. I read a book. You read.'
example = 'You read a book.'
```

```
print('Corpus:', corpus)
generate_bigram_table(corpus)

print('\nSentence:', example)
get_probability(example)
```

Output:

Corpus: You book a flight. I read a book. You read.
['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']

Bigram Probability Table:

	eos	you	book	a	flight	I	read
eos	0	0.5	0	0	0	0.25	0
you	0	0	0.5	0	0	0	0.5
book	0.5	0	0	0.5	0	0	0
a	0	0	0.5	0	0.5	0	0
flight	1	0	0	0	0	0	0
I	0	0	0	0	0	0	1
read	0.5	0	0	0.5	0	0	0

Sentence: You read a book.
Probability: you | eos = 0.5
Probability: read | you = 0.5
Probability: a | read = 0.5
Probability: book | a = 0.5
Probability: eos | book = 0.5
Probability of Sentence: 0.03125

Conclusion:

- An n-gram model is a type of probabilistic language model for predicting the next item in such a sequence in the form of a (n − 1)–order Markov model.
- n-gram models are now widely used in probability, communication theory,

computational linguistics (for instance, statistical natural language processing), computational biology (for instance, biological sequence analysis), and data compression.

- Two benefits of n-gram models (and algorithms that use them) are simplicity and scalability
- With a larger n, a model can store more context with a well-understood space–time tradeoff, enabling small experiments to scale up efficiently.

EXPERIMENT NO.4

Aim: To study Parts of Speech (POS) Tagging using Penn Tree Bank

Theory:

In corpus linguistics, part-of-speech tagging (POS tagging or POS tagging or POST), also called grammatical tagging or word-category disambiguation, is the process of marking up a word in a text (corpus) as corresponding to a particular part of speech, based on both its definition and its context—i.e., its relationship with adjacent and related words in a phrase, sentence, or paragraph. A simplified form of this is commonly taught to school-age children, in the identification of words as nouns, verbs, adjectives, adverbs, etc.

The Penn Treebank produced approximately 7 million words of part-of-speech tagged text, 3 million words of skeletally parsed text, over 2 million words of text parsed for predicate argument structure, and 1.6 million words of transcribed spoken text annotated for speech disfluencies. The material annotated includes such wide-ranging genres as IBM computer manuals, nursing notes, Wall Street Journal articles, and transcribed telephone conversations, among others.

Tags:

LS:	list item marker
TO:	"to" as preposition or infinitive marker
VBN:	verb, past participle
'':	closing quotation mark
WP:	WH-pronoun
UH:	interjection
VBG:	verb, present participle or gerund
JJ:	adjective or numeral, ordinal
VBZ:	verb, present tense, 3rd person singular
--:	dash
VBP:	verb, present tense, not 3rd person singular

NN: noun, common, singular or mass
DT: determiner
PRP: pronoun, personal
:: colon or ellipsis
WP$: WH-pronoun, possessive
NNPS: noun, proper, plural
PRP$: pronoun, possessive
WDT: WH-determiner
(: opening parenthesis
): closing parenthesis
.: sentence terminator
,: comma
``: opening quotation mark
$: dollar
RB: adverb
RBR: adverb, comparative
RBS: adverb, superlative
VBD: verb, past tense
IN: preposition or conjunction, subordinating
FW: foreign word
RP: particle
JJR: adjective, comparative
JJS: adjective, superlative
PDT: pre-determiner
MD: modal auxiliary
VB: verb, base form
WRB: Wh-adverb
NNP: noun, proper, singular
EX: existential there
NNS: noun, common, plural
SYM: symbol
CC: conjunction, coordinating
CD: numeral, cardinal
POS: genitive marker

Program:

```
import nltk
from nltk import pos_tag, word_tokenize
sents = ["Time flies like an arrow", "She promised to back the bill"]
tagdict = nltk.data.load('help/tagsets/upenn_tagset.pickle')
for sent in sents:
   tokens = word_tokenize(sent)
   print(sent)
   print(tokens)
   print()
   tags = pos_tag(tokens)
   for (word,tag) in tags:
      print(word+": ",tag)
      print(tagdict[tag][0])
      print()
```

Output:

```
Time flies like an arrow
['Time', 'flies', 'like', 'an', 'arrow']

Time:  NNP
noun, proper, singular

flies:  NNS
noun, common, plural

like:  IN
preposition or conjunction, subordinating

an:  DT
determiner

arrow:  NN
noun, common, singular or mass
```

She promised to back the bill
['She', 'promised', 'to', 'back', 'the', 'bill']

She: PRP
pronoun, personal

promised: VBD
verb, past tense

to: TO
"to" as preposition or infinitive marker

back: VB
verb, base form

the: DT
determiner

bill: NN
noun, common, singular or mass

Conclusion:

- Part of Speech Tags are useful for building parse trees, which are used in building NERs (most named entities are Nouns) and extracting relations between words.
- POS Tagging is also essential for building lemmatizers which are used to reduce a word to its root form.
- POS tagging has applications in sentiment analysis, question answering, and word sense disambiguation as well.

EXPERIMENT NO. 5

Aim: To implement Stochastic POS Tagging using Hidden Markov Model

Theory:

The HMM (Hidden Markov Model) is a sequence model. A sequence model or sequence classifier is a model whose job is to assign a label or class to each unit in a sequence, thus mapping a sequence of observations to a sequence of labels. An HMM is a probabilistic sequence model, given a sequence of units (words, letters, morphemes, sentences, whatever), it computes a probability distribution over possible sequences of labels and chooses the best label sequence.

The HMM is based on augmenting the Markov chain. A Markov chain is a model that tells us something about the probabilities of sequences of random variables, states, each of which can take on values from some set. These sets can be words, or tags, or symbols representing anything, for example the weather. A Markov chain makes a very strong assumption that if we want to predict the future in the sequence, all that matters is the current state. All the states before the current state have no impact on the future except via the current state. It's as if to predict tomorrow's weather you could examine today's weather, but you weren't allowed to look at yesterday's weather.

A Markov chain is useful when we need to compute a probability for a sequence of observable events. In many cases, however, the events we are interested in are hidden, we don't observe them directly. For example, we don't normally observe part-of-speech tags in a text. Rather, we see words, and must infer the tags from the word sequence. We call the tags hidden because they are not observed. A Hidden Markov model (HMM) allows us to talk about both observed events Hidden Markov model (like words that we see in the input) and hidden events (like part-of-speech tags) that we think of as causal factors in our probabilistic model.

Program:

```
import nltk
from nltk import word_tokenize

sentences = [
        'Mary Jane can see Will', ‘Spot will see Mary',
        'Will Jane spot Mary?', 'Mary will pat Spot' ]

taggedSentences = [
        [('mary', 'N'), ('jane', 'N'), ('can', 'M'), ('see', 'V'), ('will', 'N')],
        [('spot', 'N'), ('will', 'M'), ('see', 'V'), ('mary', 'N')],
        [('Will', 'M'), ('jane', 'N'), ('spot', 'V'), ('mary', 'N')],
        [('mary', 'N'), ('will', 'M'), ('pat', 'V'), ('spot', 'N')]
]

wordOccurence = {}
tagOccurence = {}
transmissionProbability = []
emissionProbability = []

def print_table(tableHeading, headers, itemList, probabilityList):
        print('\n' + tableHeading + '\n')
        for header in headers:
                print('\t', header, end = ' ')
        print()
        for i in range(len(itemList)):
                print(itemList[i], end = ' ')
                probabilities = probabilityList[i]
                for probability in probabilities:
                        print('\t', probability, end = ' ')
                print()

def get_occurences():
        for sentence in taggedSentences:
                for (word, tag) in sentence:
                        if word.lower() not in wordOccurence:
                                wordOccurence.update({word.lower(): 1})
                        else:
                                wordOccurence[word.lower()] += 1
                        if tag not in tagOccurence:
                                tagOccurence.update({tag: 1})
                        else:
                                tagOccurence[tag] += 1

def emission_probability():
        tagList = list(tagOccurence.keys())
        wordList = list(wordOccurence.keys())
        for word in wordList:
```

```
                wordTagCount = {'N': 0, 'M': 0, 'V': 0}
                for sentence in taggedSentences:
                        for taggedWord in sentence:
                                if taggedWord[0] == word:
                                        wordTagCount[taggedWord[1]] += 1
                emissions = [round(value / tagOccurence[tag], 2) for value, tag in
zip(wordTagCount.values(), tagList)]
                emissionProbability.append(emissions)
        print_table('Emission Probabilities:', tagList, wordList, emissionProbability)

def form_tag_pairs():
        tagPairs = []
        for sentence in taggedSentences:
                for item in range(len(sentence) - 1):
                        tagPairs.append(str(sentence[item][1] + ' ' + sentence[item+1][1]))
        return tagPairs

def count_tag_pairs(tagPairs):
        tagPairCount = {}
        for pair in tagPairs:
                if pair not in tagPairCount:
                        tagPairCount.update({pair: 1})
                else:
                        tagPairCount[pair] += 1
        return tagPairCount

def transmission_probability():
        tagList = list(tagOccurence.keys())
        tagPairs = form_tag_pairs()
        tagPairCount = count_tag_pairs(tagPairs)
        for tagA in tagList:
                transmission = []
                for tagB in tagList:
                        try:
                                value = tagPairCount[str(tagA + ' ' + tagB)] / tagOccurence[tagA]
                        except KeyError as e:
                                value = 0
                        transmission.append(round(value, 2))
                transmissionProbability.append(transmission)
        print_table('Transmission Probabilities:', tagList, tagList, transmissionProbability)

def parse_corpus(corpus):
        print ('\n\nCorpus:', corpus)
        corpusTokens = word_tokenize(corpus)
        tagList = list(tagOccurence.keys())
        emittedTags = []
        print ('\n' + 'POS Tagging:')
        print ('Word\t Tag')
        for word in range(len(corpusTokens)):
                wordIndex = list(wordOccurence.keys()).index(corpusTokens[word])
```

```
                    eProbability = emissionProbability[wordIndex]
                    maxValue = 0
                    maxTag = 0
                    if word != 0:
                            prevTag = emittedTags[word-1]
                    else:
                            prevTag = -1
                    for i in range (3):
                            probability = eProbability[i]
                            if prevTag != -1:
                                    tProbability = transmissionProbability[prevTag][i]
                            else:
                                    tProbability = 1
                            value = probability + tProbability
                            if value > maxValue:
                                    maxValue = value
                                    maxTag = i
                    emittedTags.append(maxTag)
                    tag = tagList[maxTag]
                    print('{}\t {}'.format(corpusTokens[word], tag))

get_occurences()
for sentence in sentences:
        print (sentence)
emission_probability()
transmission_probability()
corpus = 'Jane will spot Will'
parse_corpus(corpus.lower())
```

Output:

Mary Jane can see Will

Spot will see Mary

Will Jane spot Mary?

Mary will pat Spot

Emission Probabilities:

	N	M	V
mary	0.44	0.0	0.0
jane	0.22	0.0	0.0
can	0.0	0.25	0.0
see	0.0	0.0	0.5
will	0.11	0.5	0.0
spot	0.22	0.0	0.25

pat	0.0	0.0	0.25

Transmission Probabilities:

	N	M	V
N	0.11	0.33	0.11
M	0.25	0	0.75
V	1.0	0	0

Corpus: jane will spot will

POS Tagging:

Word	Tag
jane	N
will	M
spot	V
will	N

Conclusion:

- HMMs underlie the functioning of stochastic taggers and are used in various algorithms, one of the most widely used being the bi-directional inference algorithm.
- More advanced HMMs learn the probabilities not only of pairs but triples or even larger sequences. So, for example, if you've just seen a noun followed by a verb, the next item may be very likely a preposition, article, or noun, but much less likely another verb. When several ambiguous words occur together, the possibilities multiply.
- It is easy to enumerate every combination and to assign a relative probability to each one, by multiplying together the probabilities of each choice in turn. The combination with the highest probability is then chosen.

EXPERIMENT NO. 6

Aim: To study Chunking by constructing a Top Down and Bottom Up Parse Tree for a Grammar

Theory:

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.

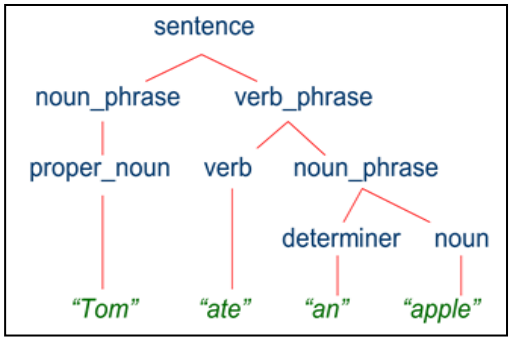
Parsing in NLP is the process of determining the syntactic structure of a text by analyzing its constituent words based on an underlying grammar (of the language).

Example sentence "Tom ate an apple". Example Grammar:

```
sentence -> noun_phrase, verb_phrase
noun_phrase -> proper_noun
noun_phrase -> determiner, noun
verb_phrase -> verb, noun_phrase
proper_noun -> [Tom]
noun -> [apple]
verb -> [ate]
determiner -> [an]
```

Then, the outcome of the parsing process would be a parse tree like the following, where sentence is the root, intermediate nodes such as noun_phrase, verb_phrase etc. have children - hence they are called non-terminals and finally, the leaves of the tree 'Tom', 'ate', 'an', 'apple' are called terminals.

Parse Tree:



Conclusion:

- 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.

EXPERIMENT NO. 7

Aim: PBLE - To import Brown Corpus, list its categories and count the number of words in the Science Fiction Category

Theory:

The Brown University Standard Corpus of Present-Day American English (or just Brown Corpus) was compiled in the 1960s by Henry Kučera and W. Nelson Francis at Brown University, Providence, Rhode Island as a general corpus (text collection) in the field of corpus linguistics.

The Corpus consists of 500 samples, distributed across 15 genres in rough proportion to the amount published in 1961 in each of those genres. All works sampled were published in 1961; as far as could be determined they were first published then and were written by native speakers of American English.

Each sample began at a random sentence-boundary in the article or other unit chosen and continued up to the first sentence boundary after 2,000 words. In a very few cases miscounts led to samples being just under 2,000 words.

Program:

```
import nltk
from nltk.corpus import brown

categories = brown.categories()
print('Categories in Brown Corpus:')
for category in categories:
   print(category)

science_fiction = brown.sents(categories = 'science_fiction')
print('Number of Sentences in Science Fiction Category:', len(science_fiction))
```

Output:

Categories in Brown Corpus:
adventure
belles_lettres
editorial
fiction
government
hobbies
humor
learned
lore
mystery
news
religion
reviews
romance
science_fiction

Number of Sentences in Science Fiction Category: 948

Conclusion:

- The Brown Corpus is an excellent learning source due to its varied coverage of works and words in domains.
- It has potential application in classification of words using this to train a Clustering algorithm.

EXPERIMENT NO. 8

Aim: PBLE - To import Brown Corpus, extract all the word tokens from the Science Fiction Category with their tags, and calculate a Frequency Distribution of the tags.

Theory:

The Brown University Standard Corpus of Present-Day American English (or just Brown Corpus) was compiled in the 1960s by Henry Kučera and W. Nelson Francis at Brown University, Providence, Rhode Island as a general corpus (text collection) in the field of corpus linguistics.

It contains 500 samples of English-language text, totaling roughly one million words, compiled from works published in the United States in 1961. The tagged Brown Corpus was formed later and used a selection of about 80 parts of speech, as well as special indicators for compound forms, contractions, foreign words and a few other phenomena, and formed the basis for many later corpora.

Program:

```
import nltk
from nltk.corpus import brown

bsf = brown.tagged_words(categories = 'science_fiction')

tagList = {}
for (word, tag) in bsf:
   if tag not in tagList:
      tagList.update({tag: 1})
   else:
      tagList[tag] += 1

tagList = sorted(tagList.items(), key = lambda item: (item[1], item[0]), reverse = True)

print('Tag\t\tCount')
for (tag, count) in tagList:
   print('{}\t\t{}'.format(tag, count))
```

Output:

Tag	Count
NN	1541
IN	1176
.	1077
AT	1040
,	791
JJ	723
NNS	532
VBD	531
RB	522
VB	495
CC	415
NP	387
PPS	336
VBN	318
PPSS	282
PP$	272
CS	269
PPO	252
``	235
''	235
VBG	203
BEDZ	200
TO	192
MD	192
HVD	135
DT	118
AP	112
NN-TL	108
QL	105
RP	100
*	95
BE	80
WDT	78
WRB	75

CD	75
ABN	65
HV	61
BED	59
--	52
BEZ	50
PN	48
BEN	40
DTI	39
NP-TL	38
DOD	38
EX	37
JJ-TL	36
WPS	28
BER	28
VBZ	26
PPL	26
NNS-TL	26
DO	26
UH	25
JJR	25
NP$	23
DTS	22
RBR	21
OD	19
NPS	18
MD*	18
DO*	13
PPSS+BEM	12
NN$	12
'	11
PPSS+HV	10
:	10
QLP	9
HVZ	9

BEG	9
PPSS+MD	8
JJT	8
HVN	8
)	8
(	8
EX+BEZ	7
BEZ*	7
BEM	7
PPSS+BER	6
IN-TL	6
PPS+BEZ	5
PP$$	5
NR	5
DOD*	5
NN$-TL	4
HVG	4
DTX	4
DT+BEZ	4
CD-HL	4
BEDZ*	4
AT-TL	4
ABL	4
WP$	3
WDT+BEZ	3
VBN-TL	3
PPLS	3
DOZ*	3
DOZ	3
AP-TL	3
WPO	2
VB+PPO	2
PPS+MD	2
OD-TL	2
NN-HL	2

FW-VB	2
FW-NNS	2
FW-NN	2
FW-AT+NN-TL	2
ABX	2
WPS+MD	1
RBT	1
PPSS+HVD	1
NPS$	1
NNS$-TL	1
NNS$	1
JJS	1
FW-NR-TL	1
FW-NNS-TL	1
FW-JJ-TL	1
FW-IN-TL	1
FW-CC-TL	1
FW-CC	1
CC-TL	1
BER*	1
BED*	1
AP$	1
.-HL	1

Conclusion:

An interesting result of the Corpus is that even for large samples, graphing words in order of decreasing frequency of occurrence shows a hyperbola: the frequency of the n-th most frequent word is roughly proportional to 1/n.