Experiment No. 1

Title:

Preprocessing of Text (Tokenization, Filtration, Script validation, Stop Word Removal, Stemming)

Aim:

The aim of this practical is to perform text preprocessing, including:

1. Tokenization
2. Filtration
3. Script Validation
4. Stop Word Removal
5. Stemming.

Lab Objectives:

To implement and analyze the basics of preprocessing text.

Theory:

In Natural Language Processing (NLP), the preprocessing of text is a crucial step to clean and prepare raw text data for further analysis.

It involves several key techniques:

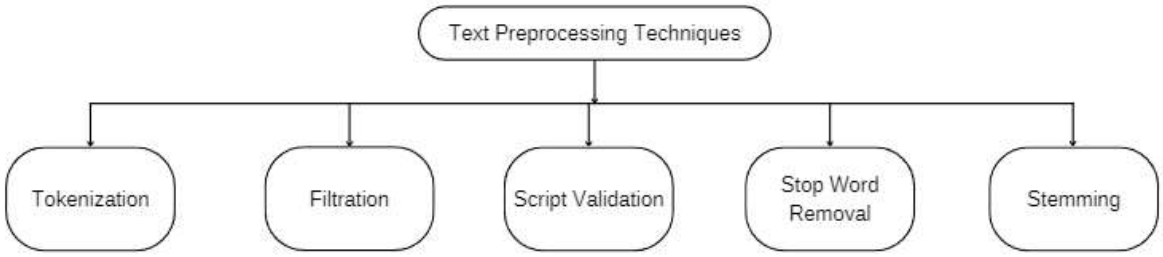


Fig 1.1 - Techniques of Text Processing

Here's an explanation of each:

1. **Tokenization:**

   Tokenization is the process of breaking a text into individual words or tokens. These tokens are the basic units of meaning in a text.

   For example, the sentence "The quick brown fox" would be tokenized into ["The", "quick", "brown", "fox"].

   Tokenization helps in converting unstructured text into a structured format that can be used for various NLP tasks like sentiment analysis, language modeling, etc.

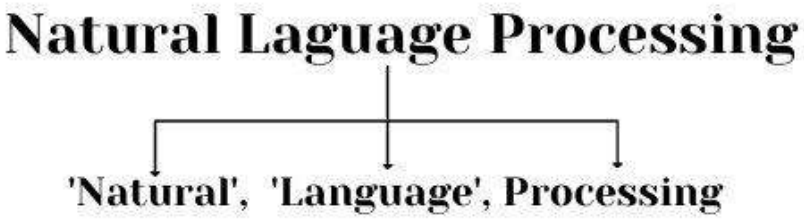


Fig 1.2 - Example of Tokenization

2. **Filtration:**

   Filtration involves removing unwanted or irrelevant characters or symbols from the text. This may include special characters, punctuation marks, or any non-alphabetic characters. Filtration helps in simplifying the text and reducing noise, making it easier for subsequent processing steps.

3. **Script Validation:**

   Script validation involves ensuring that the text is in the expected script or writing system. For instance, if you're working with English text, you'd want to check that the text is indeed in the Roman script. This step helps avoid potential issues that may arise from mixing different scripts in a text corpus.

4. **Stop Word Removal:**

   Stop words are common words (e.g., "the", "is", and "in") that don't carry significant meaning on their own. Removing them can help reduce the dimensionality of the data and focus on more important terms. Stop word removal can improve the efficiency of NLP algorithms, as it reduces the computational load and noise in the data.

```
print(stopwords)
```

```
['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', "you're", "you've", "you'll", "you'd", 'yo
ur', 'yours', 'yourself', 'yourselves', 'he', 'him', 'his', 'himself', 'she', "she's", 'her', 'hers', 'herself
', 'it', "it's", 'its', 'itself', 'they', 'them', 'their', 'theirs', 'themselves', 'what', 'which', 'who', 'who
m', 'this', 'that', "that'll", 'these', 'those', 'am', 'is', 'are', 'was', 'were', 'be', 'been', 'being', 'have
', 'has', 'had', 'having', 'do', 'does', 'did', 'doing', 'a', 'an', 'the', 'and', 'but', 'if', 'or', 'because',
'as', 'until', 'while', 'of', 'at', 'by', 'for', 'with', 'about', 'against', 'between', 'into', 'through', 'dur
ing', 'before', 'after', 'above', 'below', 'to', 'from', 'up', 'down', 'in', 'out', 'on', 'off', 'over', 'under
', 'again', 'further', 'then', 'once', 'here', 'there', 'when', 'where', 'why', 'how', 'all', 'any', 'both', 'e
ach', 'few', 'more', 'most', 'other', 'some', 'such', 'no', 'nor', 'not', 'only', 'own', 'same', 'so', 'than',
'too', 'very', 's', 't', 'can', 'will', 'just', 'don', "don't", 'should', "should've", 'now', 'd', 'll', 'm', '
o', 're', 've', 'y', 'ain', 'aren', "aren't", 'couldn', "couldn't", 'didn', "didn't", 'doesn', "doesn't", 'hadn
', "hadn't", 'hasn', "hasn't", 'haven', "haven't", 'isn', "isn't", 'ma', 'mightn', "mightn't", 'mustn', "mustn'
t", 'needn', "needn't", 'shan', "shan't", 'shouldn', "shouldn't", 'wasn', "wasn't", 'weren', "weren't", 'won',
"won't", 'wouldn', "wouldn't"]
```

Fig 1.3 - List of Stop Words

5. Stemming:

Stemming is the process of reducing a word to its base or root form. This is achieved by removing suffixes or prefixes. For example, "running" and "ran" would both be stemmed to "run". Stemming helps in reducing the variations of words to a common base form. This can be particularly useful for tasks like document retrieval or text classification.

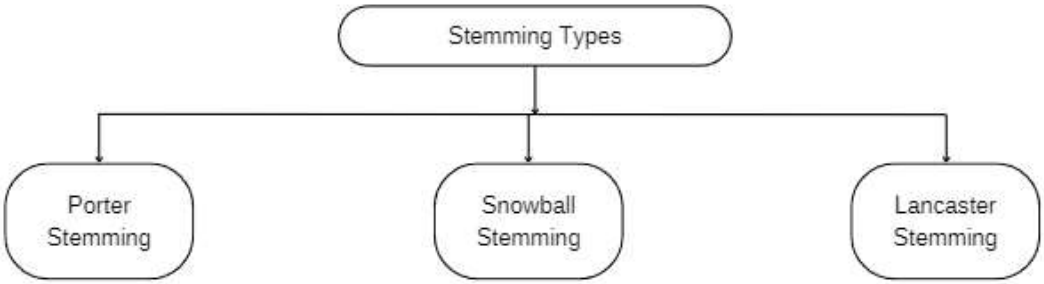


Fig 1.4 - Types of Stemming

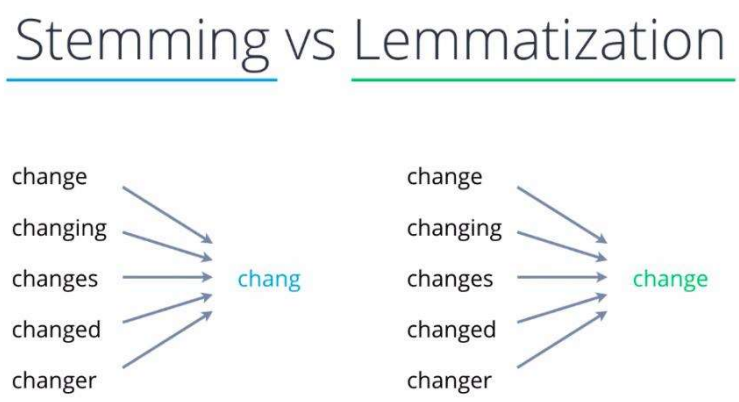


Fig 1.5 - Difference of Stemming & Lemmatization

Examples:

Consider the sentence: "The quick brown fox jumps over the lazy dog."

After applying the preprocessing steps:

- Tokenization: ["The", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog"]
- Filtration: ["The", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog"]
- Script Validation: English script (Roman alphabet)
- Stop Word Removal: ["quick", "brown", "fox", "jumps", "lazy", "dog"]
- Stemming: ["quick", "brown", "fox", "jump", "lazi", "dog"]

Algorithm:

Step 1: Importing Libraries

- import pandas
- import string
- from nltk.corpus import stopwords
- import re
- from nltk.stem.porter import PorterStemmer
- from nltk.stem import WordNetLemmatizer
- import numpy
- from pprint import print

Step 2: Data Loading and Cleaning

- Load the dataset "Twitter Sentiments.csv" into a pandas DataFrame ('df').
- Drop unnecessary columns 'id' and 'label'.
- Convert the text in the 'tweet' column to lowercase and store it in a new column 'clean text'.

Step 3: Remove Punctuation

- Define a function 'remove punct(text)' that takes a text input and removes any punctuation.
- Apply this function to the 'clean text' column and store the result in a new column 'punct text'.

Step 4: Remove Stopwords

- Define a function 'remove stopwords(text)' that takes a text input and removes stopwords.
- Apply this function to the 'punct text' column and store the result in a new column 'stopwords text'.

Step 5: Tokenization

- Define a function 'tokenization(text)' that splits the text into tokens.
- Apply this function to the 'stopwords text' column and store the result in a new column 'tokenized text'.

Step 6: Stemming

- Define a function 'stemming(text)' that applies stemming using a Porter Stemmer.
- Apply this function to the 'tokenized text' column and store the result in a new column 'stemmed text'.

Step 7: Lemmatization

- Define a function 'lemmatizer(text)' that applies lemmatization using a WordNet Lemmatizer.
- Apply this function to the 'stemmed text' column and store the result in a new column 'lemmatized text'.

Step 8: Vocabulary Building and Validation

- Define a function 'tokenize(text)' that splits the text into tokens.
- Iterate through the training data, tokenize each document, and build a vocabulary of unique tokens.
- Define a function 'validate in vocabulary(text)' that checks if at most 10 percent of words in a text are unknown. Raise a 'ValidationError' if this condition is not met.

Step 9: Schema Definition and Validation

- Define a schema 'TweetSchema' that specifies the structure of tweet data, including validation rules.
- Prepare test data in the required format (list of dictionaries with 'id' and 'text').
- Attempt to load the test data using the defined schema. If a 'ValidationError' occurs, print the error messages and exclude the invalid rows from the result.

Step 10: Execution

- Load the train and test datasets from "train.csv" and "test.csv". Install the marshmallow library if it's not already installed.
- Execute the schema validation process and print the number of received rows, error logs (if any), and the final count of valid rows.

Flowchart:

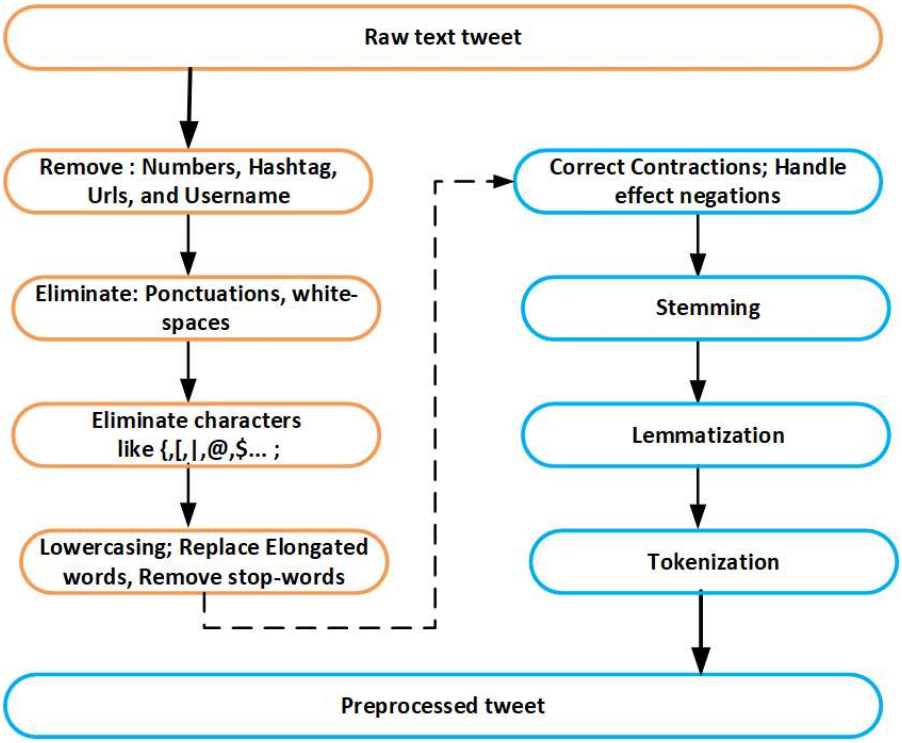


Fig. 1.6 - Flowchart of Text Processing Program

Lab Outcome:

Implement and analyze the basics of preprocessing text.

Conclusions:

In conclusion, text preprocessing plays a pivotal role in enhancing the efficiency and accuracy of natural language processing tasks. Tokenization facilitates the decomposition of text into meaningful units, filtration ensures the removal of irrelevant characters and noise, script validation maintains consistency, stop-word removal eliminates redundant terms, and stemming reduces words to their root forms. This comprehensive preprocessing pipeline not only optimizes computational resources but also contributes to the overall effectiveness of text analysis and machine learning applications, enabling more robust and insightful outcomes from textual data.

Program:

```
# Step-1
import pandas as pd
import string
from nltk.corpus import stopwords
import re
from nltk.stem.porter import PorterStemmer
from nltk.stem import WordNetLemmatizer
import numpy as np
from pprint import print

# Step-2
df = pd.read_csv("Twitter Sentiments.csv")
df.head()
df = df.drop(columns=['id','label'], axis=1)
df.head()
df['clean_text'] = df['tweet'].str.lower()
df.head()
```

```
# Step-3
string.punctuation
def remove_punct(text):
punctuations = string.punctuation
return text.translate(str.maketrans(”,”,punctuations))
df[’punct_text’] = df[’clean_text’].apply(lambda x: remove_punct(x))
df.head()

# Step-4
” ”.join(stopwords.words(’english’))
STOPWORDS = set(stopwords.words(’english’))
def remove_stopwords(text):
return ” ”.join([word for word in text.split() if word not in STOPWORDS])
df[’stopwords_txt’] = df[’punct_text’].apply(lambda x: remove_stopwords(x))
df.head()

# Step-5
def tokenization(text):
tokens = re.split(’W+’,text)
return tokens
df[’tokenized_text’] = df[’stopwords_txt’].apply(lambda x: tokenization(x))
df.head()

# Step-6
porter_stemmer = PorterStemmer()
def stemming(text):
stem_text = [porter_stemmer.stem(word) for word in text]
return stem_text
df[’stemmed_text’] = df[’tokenized_text’].apply(lambda x: stemming(x))
df.head()

# Step-7
wordnet_lemmatizer = WordNetLemmatizer()
def lemmatizer(text):
lemm_text = [wordnet_lemmatizer.lemmatize(word) for word in text]
return lemm_text
df[’lemmatize_text’] = df[’stemmed_text’].apply(lambda x: lemmatizer(x))
df.head()
```

```
# Step-8
train = pd.read_csv(”train.csv”)
test = pd.read_csv(”test.csv”)
from marshmallow import Schema, fields, validate, ValidationError
from typing import List
pip install -U marshmallow
def tokenize(text: str) -> List[str]:
”””Split the given text into tokens by splitting at whitespace.”””
return text.split(” ”)
vocabulary = list()
for doc in train.text:
for token in tokenize(doc):
if token not in vocabulary:
vocabulary.append(token)
vocabulary = set(vocabulary)
def validate_in_vocabulary(text: str) -> bool:
”””” Only allow for at most 10 percent of unknown words in the text. Raise ValidationError and print the unknown words. ”””””
tokenized_text = tokenize(text)
unknown_words = [t for t in tokenized_text if t not in vocabulary]
known_word_score = (len(tokenized_text) - len(unknown_words)) / len(tokenized_text)
if known_word_score < 0.1 and len(unknown_words) > 0:
raise ValidationError(”To many unknown words: ”.format(unknown_words))

# Step-9
class TweetSchema(Schema):
id = fields.Integer(required=True)
text = fields.String(validate=[ validate_in_vocabulary, validate.Length(min=7), validate.Length(max=280) ])

# Step-10
test_data = test[[”id”, ”text”]].to_dict(orient=”records”)
try:
print(” Received rows.”.format(len(test_data)))
result = TweetSchema(many=True).load(test_data)
except ValidationError as err:
print(”Error log:”)
pprint(err.messages)
result = [d for i, d in enumerate(err.valid_data) if i not in err.messages.keys()]
finally: print(” Received valid rows, removed rows” .format(len(result), len(test_data) - len(result)))
```

Program Output Screenshot:

Out[25]:

	tweet	clean_text	punct_text	stopwords_text	tokenized_text
0	@user when a father is dysfunctional and is s...	@user when a father is dysfunctional and is s...	user when a father is dysfunctional and is so...	user,father,dysfunctional,selfish,drags,kids,d...	[user,father,dysfunctional,selfish,drags,kids,...
1	@user @user thanks for #lyft credit i can't us...	@user @user thanks for #lyft credit i can't us...	user user thanks for lyft credit i cant use ca...	user,user,thanks,lyft,credit,cant,use,cause,do...	[user,user,thanks,lyft,credit,cant,use,cause,d...
2	bihday your majesty	bihday your majesty	bihday your majesty	bihday,majesty	[bihday,majesty]
3	#model i love u take with u all the time in ...	#model i love u take with u all the time in ...	model i love u take with u all the time in u...	model,love,u,take,u,time,urð□□±,ð□□□ð□□□ð□□□ð□...	[model,love,u,take,u,time,urð□□±,ð□□□ð□□□ð□□□ð...
4	factsguide: society now #motivation	factsguide: society now #motivation	factsguide society now motivation	factsguide,society,motivation	[factsguide,society,motivation]

stemmed_text	lemmatize_text
[user,father,dysfunctional,selfish,drags,kids,...	[user,father,dysfunctional,selfish,drags,kids,...
[user,user,thanks,lyft,credit,cant,use,cause,d...	[user,user,thanks,lyft,credit,cant,use,cause,d...
[bihday,majesti]	[bihday,majesti]
[model,love,u,take,u,time,urð□□±,ð□□□ð□□□ð□□□ð...	[model,love,u,take,u,time,urð□□±,ð□□□ð□□□ð□□□ð...
[factsguide,society,motiv]	[factsguide,society,motiv]

In [31]:
```
import nltk
nltk.download('punkt')
nltk.download('stopwords')
```

```
[nltk_data] Downloading package punkt to
[nltk_data]     C:\Users\shrey\AppData\Roaming\nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package stopwords to
[nltk_data]     C:\Users\shrey\AppData\Roaming\nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

Out[31]:
```
True
```

In [7]:
```python
#TOKENIZATION

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

# Sample text
text = """
Despite the fact that piranhas are relatively harmless, many people continue to
"""

# Tokenize sentences
sentences = sent_tokenize(text)
print("Sentence Tokenization:")
print(sentences)

# Tokenize words
words = word_tokenize(text)
print("\nWord Tokenization:")
print(words)
```

```
Sentence Tokenization:
['\nDespite the fact that piranhas are relatively harmless, many people conti
nue to believe the pervasive myth that piranhas are dangerous to humans.', 'T
his impression of piranhas is exacerbated by their mischaracterization in pop
ular media.', 'For instance, the promotional poster for the 1978 horror film
Piranha features an oversized piranha poised to bite the leg of an unsuspecti
ng woman.', 'Such a terrifying representation easily captures the imagination
and promotes unnecessary fear.', 'While the trope of the man-eating piranhas
lends excitement to adventure stories, it bears little resemblance to the rea
l-life piranha.', 'By paying more attention to fact than fiction, humans may
finally be able to let go of this inaccurate belief.']

Word Tokenization:
['Despite', 'the', 'fact', 'that', 'piranhas', 'are', 'relatively', 'harmles
s', ',', 'many', 'people', 'continue', 'to', 'believe', 'the', 'pervasive',
'myth', 'that', 'piranhas', 'are', 'dangerous', 'to', 'humans', '.', 'This',
'impression', 'of', 'piranhas', 'is', 'exacerbated', 'by', 'their', 'mischara
cterization', 'in', 'popular', 'media', '.', 'For', 'instance', ',', 'the',
'promotional', 'poster', 'for', 'the', '1978', 'horror', 'film', 'Piranha',
'features', 'an', 'oversized', 'piranha', 'poised', 'to', 'bite', 'the', 'le
g', 'of', 'an', 'unsuspecting', 'woman', '.', 'Such', 'a', 'terrifying', 'rep
resentation', 'easily', 'captures', 'the', 'imagination', 'and', 'promotes',
'unnecessary', 'fear', '.', 'While', 'the', 'trope', 'of', 'the', 'man-eatin
g', 'piranhas', 'lends', 'excitement', 'to', 'adventure', 'stories', ',', 'i
t', 'bears', 'little', 'resemblance', 'to', 'the', 'real-life', 'piranha',
'.', 'By', 'paying', 'more', 'attention', 'to', 'fact', 'than', 'fiction',
',', 'humans', 'may', 'finally', 'be', 'able', 'to', 'let', 'go', 'of', 'thi
s', 'inaccurate', 'belief', '.']
```

In [13]:
```
#FILTERATION

# Get English stopwords
stop_words = set(stopwords.words('english'))

# Filter out stop words and punctuation
filtered_words = [word for word in words if word.lower() not in stop_words and

print("Filtered Words:")
print(filtered_words)
```

```
Filtered Words:
['Despite', 'fact', 'piranhas', 'relatively', 'harmless', 'many', 'people',
'continue', 'believe', 'pervasive', 'myth', 'piranhas', 'dangerous', 'human
s', 'impression', 'piranhas', 'exacerbated', 'mischaracterization', 'popula
r', 'media', 'instance', 'promotional', 'poster', '1978', 'horror', 'film',
'Piranha', 'features', 'oversized', 'piranha', 'poised', 'bite', 'leg', 'unsu
specting', 'woman', 'terrifying', 'representation', 'easily', 'captures', 'im
agination', 'promotes', 'unnecessary', 'fear', 'trope', 'man-eating', 'piranh
as', 'lends', 'excitement', 'adventure', 'stories', 'bears', 'little', 'resem
blance', 'real-life', 'piranha', 'paying', 'attention', 'fact', 'fiction', 'h
umans', 'may', 'finally', 'able', 'let', 'go', 'inaccurate', 'belief']
```

In [19]:
```
#SCRIPT VALIDATION

import py_compile

# Validate the script for syntax errors
try:
    py_compile.compile('C:/Users/shrey/Untitled1.py', doraise=True)
    print("Syntax is valid.")
except py_compile.PyCompileError as e:
    print("Syntax error found:")
    print(e)
except FileNotFoundError:
    print("Syntax is valid")
```

```
Syntax is valid
```

In [21]:
```python
#UPPERCASE CONVERSION

# Tokenize the text into words
words = nltk.word_tokenize(text)

# Get English stopwords
stop_words = set(stopwords.words('english'))

# Filter out stop words and punctuation
filtered_words = [word for word in words if word.lower() not in stop_words and

# Convert the filtered words to uppercase
filtered_uppercase_words = [word.upper() for word in filtered_words]

# Join the words back into a single string
result_text = ' '.join(filtered_uppercase_words)

print(result_text)
```

```
DESPITE FACT PIRANHAS RELATIVELY HARMLESS MANY PEOPLE CONTINUE BELIEVE PERVAS
IVE MYTH PIRANHAS DANGEROUS HUMANS IMPRESSION PIRANHAS EXACERBATED MISCHARACT
ERIZATION POPULAR MEDIA INSTANCE PROMOTIONAL POSTER 1978 HORROR FILM PIRANHA
FEATURES OVERSIZED PIRANHA POISED BITE LEG UNSUSPECTING WOMAN TERRIFYING REPR
ESENTATION EASILY CAPTURES IMAGINATION PROMOTES UNNECESSARY FEAR TROPE MAN-EA
TING PIRANHAS LENDS EXCITEMENT ADVENTURE STORIES BEARS LITTLE RESEMBLANCE REA
L-LIFE PIRANHA PAYING ATTENTION FACT FICTION HUMANS MAY FINALLY ABLE LET GO I
NACCURATE BELIEF
```

In [27]:
```python
#STEMMING

from nltk.stem import PorterStemmer

# Create a Porter Stemmer instance
porter_stemmer = PorterStemmer()

# Example words for stemming
words = ["running", "jumps", "happily", "running", "happily"]

# Apply stemming to each word
stemmed_words = [porter_stemmer.stem(word) for word in words]

# Print the results
print("Original words:", words)
print("Stemmed words:", stemmed_words)
```

```
Original words: ['running', 'jumps', 'happily', 'running', 'happily']
Stemmed words: ['run', 'jump', 'happili', 'run', 'happili']
```

In [29]:
```
#LEMMATIZATION

# import these modules
from nltk.stem import WordNetLemmatizer

lemmatizer = WordNetLemmatizer()

print("rocks :", lemmatizer.lemmatize("rocks"))
print("corpora :", lemmatizer.lemmatize("corpora"))

# a denotes adjective in "pos"
print("better :", lemmatizer.lemmatize("better", pos="a"))
```

```
rocks : rock
corpora : corpus
better : good
```

In [ ]:

Experiment No. 2

Title:

Morphological Analysis

Aim:

The aim of the practical is to conduct morphological analysis to understand and analyze the structure of linguistic forms.

Lab Objectives:

To apply and demonstrate the basics of morphological analysis for word formation.

Theory:

Morphology analysis is a linguistic study that deals with the internal structure of words in a language. It explores how words are formed from smaller meaningful units called morphemes. Morphemes are the smallest grammatical units in a language that carry meaning.

Here are the key components and concepts of morphology analysis:

Morpheme:

As mentioned earlier, a morpheme is the smallest unit of meaning in a language. It can be a word itself or a part of a word (prefixes, suffixes, etc.). For example, in the word ”unhappiness,” there are three morphemes: ”un-” (meaning ’not’), ”happy” (meaning ’joyful’), and ”-ness” (indicating a state or quality).

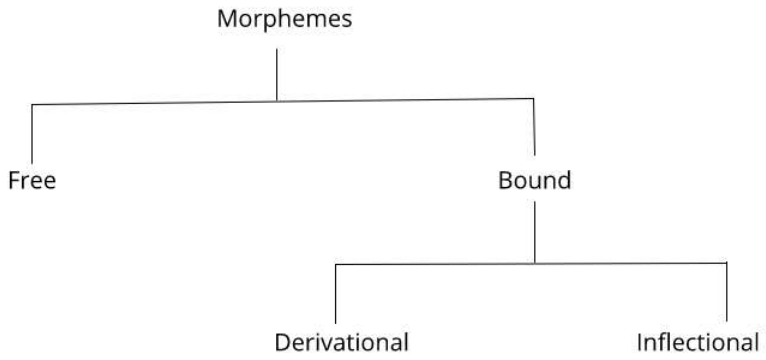


Fig. 2.1 - Types of Morphemes

Free and Bound Morphemes:

Free morphemes can stand alone as words by themselves and carry meaning. For example, in English, words like ”dog,” ”run,” and ”book” are free morphemes.

Bound morphemes can't stand alone and must be attached to other morphemes to convey meaning. Prefixes like ”un-” or suffixes like ”-ed” are examples.

Inflectional vs. Derivational Morphemes:

Inflectional morphemes alter the form of a word to indicate grammatical information like tense, case, number, etc. For example, in English, ”-s” in ”cats” indicates plural. Derivational morphemes are used to create new words or change the grammatical category of a word.

For example, adding ”-ness” to ”happy” creates ”happiness,” changing an adjective to a noun.

Morphological Processes:

Affixation: This involves adding prefixes (before the root) or suffixes (after the root) to a word. For example, adding ”-ing” to ”read” forms ”reading.”

Compounding: It involves combining two or more free morphemes to create a new word. For instance, ”blackboard” combines ”black” and ”board.”

Reduplication: This process involves repeating a morpheme or part of it to create a new word. For example, in Tagalog, ”tawa” (laugh) becomes ”tatawa” (will laugh).

Blending: This involves combining parts of two words to create a new one, often through truncation or fusion. For example, ”brunch” combines ”breakfast” and ”lunch.”

Conversion (Zero Derivation): This is a process where a word changes its grammatical category without any overt morphological change. For example, ”to google” (verb) from the proper noun ”Google.”

Allomorphy: This refers to the variation in pronunciation of a morpheme depending on its context. For example, the plural morpheme in English can be pronounced as /s/ (cats) or /z/ (dogs) depending on the preceding sound.

Examples:

Word: Unhappiness

Morphemes:

”un-” (bound morpheme)

”happi-” (free morpheme)

”-ness” (bound morpheme)

Affixes:

Prefix: ”un-”

Suffix: ”-ness”

Inflectional vs. Derivational Morphemes:

Inflectional: None in this example.

Derivational: ”un-” and ”-ness” are derivational morphemes because they create a new word and change its grammatical category.

Algorithm:

Here's a step-by-step algorithm:

Step-1 Import necessary libraries

- Import the spaCy library for natural language processing.
- Import the 'tabulate' module for creating tables.

Step-2: Load the spaCy model

- Load the English model using 'spacy.load()' and assign it to the variable 'nlp'.

Step-3: Get user input

- Prompt the user to enter an interrogative, declarative, and complex sentence.

Step-4: Process the sentences with spaCy

- Use the 'nlp' object to process the sentences and create Doc objects.

Step-5: Extract POS information

- For each sentence, extract the text and its corresponding part of speech (POS) tag using a list comprehension.

Step-6: Define table headers

- Create a list of headers for the tables.

Step-7: Create tables

- Use the 'tabulate' function to format the POS information into tables.

Step-8: Print the tables

- Display the tables with the POS information.

Step-9: End of code

Flowchart:

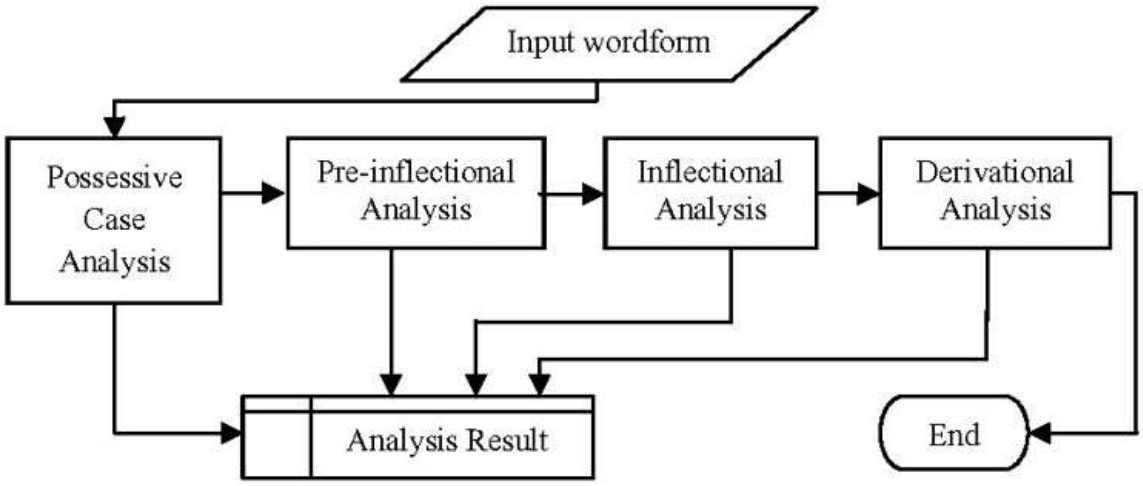


Fig. 2.2 - Flowchart of Morphological Analysis Program

Lab Outcome:

Apply and demonstrate the basics of morphological analysis for word formation.

Conclusions:

In conclusion, Morphological Analysis proves its worth as a valuable and versatile tool with applications in diverse fields. Its systematic approach to breaking down complex systems facilitates comprehensive understanding, aiding in problem-solving, decision-making, and system optimization. By dissecting the morphology of entities, this method offers a structured approach, fostering creativity and efficiency in engineering, decision science, strategic planning, and beyond. Its adaptability makes Morphological Analysis a valuable asset for addressing multifaceted challenges across various disciplines.

Program:

```
# Step-1
import spacy
from tabulate import tabulate
# Step-2
nlp = spacy.load(”en_core_web_sm”)
# Step-3
interrogative_sentence = input(”Enter an interrogative sentence: ”)
declarative_sentence = input(”Enter a declarative sentence: ”)
complex_sentence = input(”Enter a complex sentence: ”)
# Step-4
interrogative_doc = nlp(interrogative_sentence)
declarative_doc = nlp(declarative_sentence)
complex_doc = nlp(complex_sentence)
```

```
# Step-5
```

```
interrogative_pos = [(token.text, token.pos_) for token in interrogative_doc]
declarative_pos = [(token.text, token.pos_) for token in declarative_doc]
complex_pos = [(token.text, token.pos_) for token in complex_doc]
# Step-6
table_headers = [”Word”, ”POS Tag”]
# Step-7
interrogative_table = tabulate(interrogative_pos, headers=table_headers, tablefmt=”pretty”)
declarative_table = tabulate(declarative_pos, headers=table_headers, tablefmt=”pretty”)
complex_table = tabulate(complex_pos, headers=table_headers, tablefmt=”pretty”)
# Step-8
print(”Interrogative Sentence POS Tags:”)
print(interrogative_table)
print(”Declarative Sentence POS Tags:”)
print(declarative_table)
print(”Complex Sentence POS Tags:”)
print(complex_table)
```

Program Output Screenshot:

```
Enter an interrogative sentence: What is the weather like today?
Enter a declarative sentence: The weather is sunny.
Enter a complex sentence: I went to the store, but they were closed, so I had to go to another store.

Interrogative Sentence POS Tags:
+---------+---------+
|  Word   | POS Tag |
+---------+---------+
|  What   |  PRON   |
|   is    |   AUX   |
|   the   |   DET   |
| weather |  NOUN   |
|  like   |   ADP   |
|  today  |  NOUN   |
|    ?    |  PUNCT  |
+---------+---------+

Declarative Sentence POS Tags:
+---------+---------+
|  Word   | POS Tag |
+---------+---------+
|   The   |   DET   |
| weather |  NOUN   |
|   is    |   AUX   |
|  sunny  |   ADJ   |
|    .    |  PUNCT  |
+---------+---------+
```

```
Complex Sentence POS Tags:
+---------+---------+
|  Word   | POS Tag |
+---------+---------+
|    I    |  PRON   |
|  went   |  VERB   |
|   to    |   ADP   |
|   the   |   DET   |
|  store  |  NOUN   |
|    ,    |  PUNCT  |
|   but   |  CCONJ  |
|  they   |  PRON   |
|  were   |   AUX   |
| closed  |  VERB   |
|    ,    |  PUNCT  |
|   so    |  CCONJ  |
|    I    |  PRON   |
|   had   |  VERB   |
|   to    |  PART   |
|   go    |  VERB   |
|   to    |   ADP   |
| another |   DET   |
|  store  |  NOUN   |
|    .    |  PUNCT  |
+---------+---------+
```

Experiment No. 3

Title:

NGram Model.

Aim:

The aim of the practical is to implement and analyze an NGram Model for language modeling.

Lab Objectives:

To construct N-gram from a given corpus and calculate the probability of a sentence, as well as identify and classify named entities in a text into predefined categories.

Theory:

N-grams are an essential concept in Natural Language Processing (NLP) used to analyze and model sequences of words or characters in text. They play a crucial role in various NLP tasks such as language modeling, text generation, machine translation, and speech recognition. Here's a detailed explanation of N-grams:

Definition:

"An N-gram is a contiguous sequence of N items (which can be words, characters, or other units) from a given sample of text or speech. These items can be letters, syllables, words, or even longer sequences, depending on the application. N-grams are used to capture the local linguistic context within a text."

Types of N-grams:

Unigrams (1-grams): These are single words. For example, in the sentence "I love NLP," the unigrams are "I," "love," and "NLP."

Bigrams (2-grams): These consist of pairs of consecutive words. In the same sentence, examples of bigrams are "I love" and "love NLP."

Trigrams (3-grams): These are sequences of three consecutive words, such as "I love NLP."

4-grams, 5-grams, etc.: You can have N-grams of any length, depending on your specific application.

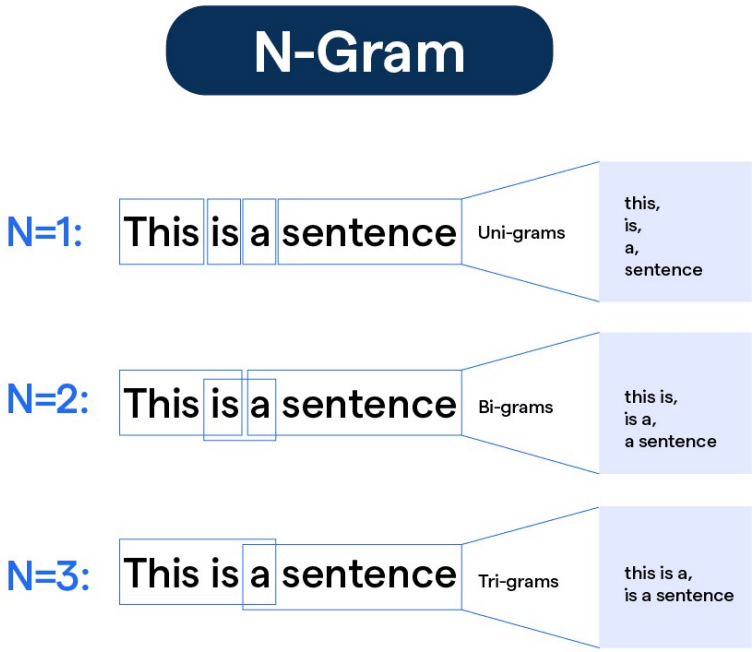


Fig. 3.1 - Examples on 3 types of N-Grams

Uses of N-grams:

- **Language Modeling:** N-grams are used to estimate the probability of a word given its previous N-1 words. This is fundamental in predicting the next word in a sequence of text, making them essential for applications like auto-completion and text generation.
- **Text Classification:** N-grams can be used to represent documents or text for classification tasks. By counting the occurrences of N-grams in a document, it's possible to create a feature vector that can be used in machine learning algorithms for text classification.
- **Information Retrieval:** In search engines, N-grams can be used to index documents and query terms. This helps in ranking and retrieving relevant documents.
- **Speech Recognition:** N-grams can be used to model sequences of phonemes or words in speech recognition systems, aiding in accurate transcription of spoken language.
- **Machine Translation:** N-grams are used in machine translation to align and translate sequences of words or phrases in different languages.
- **Spelling Correction:** N-grams can help identify and correct misspelled words by comparing them to correctly spelled N-grams in a language model.

Limitations:

- **Sparsity:** For larger values of N, you might encounter the data sparsity problem. Many N-grams may occur very infrequently or not at all in a given dataset, which can make them less reliable for modeling.
- **Loss of Context:** N-grams have limited contextual information, especially for larger values of N. They can't capture long-range dependencies in text.
- **Smoothing:** To address the sparsity issue, techniques like Laplace (add-one) smoothing or Good-Turing smoothing are often used to estimate the probabilities of unseen N-grams based on the observed ones.

Backoff and Interpolation:

For better language modeling, you can combine N-grams of different lengths using techniques like backoff and interpolation to capture various levels of context.

N-gram Probability:

The probability of an n-gram occurring in a text can be estimated using the following formula:

$$P(w_n|w_{n-1}, w_{n-2}, \ldots, w_1) = \frac{Count(w_{n-1}, w_{n-2}, \ldots, w_1, w_n)}{Count(w_{n-1}, w_{n-2}, \ldots, w_1)}$$

Where:

w_n is the nth word.

$w_{n-1}, w_{n-2}, \ldots, w_1$ are the previous words in the sequence.

$Count(\cdot)$ represents the frequency of the specified sequence in the corpus.

Fig. 3.2 - Formula of N-gram Probability

Examples:

Consider the sentence: "The quick brown fox jumps over the lazy dog."

Unigrams (N=1): "The", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog."

Bigrams (N=2): "The quick", "quick brown", "brown fox", "fox jumps", "jumps over", "over the", "the lazy", "lazy dog."

Trigrams (N=3): "The quick brown", "quick brown fox", "brown fox jumps", "fox jumps over", "jumps over the", "over the lazy", "the lazy dog."

Algorithm:

Here's an algorithm to explain the code step by step:

Step-1: Import Necessary Libraries

- Import the defaultdict class from the collection's module
- The random module.

Step-2: Define the Function generate_ngrams

Input:

- text: A string containing the input text.
- n: An integer representing the 'N' in N-grams.

Output:

- ngrams: A list of tuples, where each tuple is an N-gram.

Steps:

1. Split the input text into a list of words.
2. Generate N-grams by creating tuples of 'n' consecutive words.
3. Return the list of N-grams.

Step-3: Define the Function calculate_probability_matrix

Input:

- ngrams: A list of tuples representing N-grams.

Output:

- probabilities: A dictionary containing probabilities.

Steps:

1. Initialize a defaultdict named counts to store counts of prefixes and their corresponding suffixes.
2. Iterate over each N-gram.
3. Extract the prefix (all elements except the last one) and the suffix (the last element).
4. Increment the count of that suffix for the given prefix.
5. Calculate probabilities based on the counts.

6. Return the probabilities.

Step-4: Define the Function print_ngram_probability

Input:

- ngrams: A list of tuples representing N-grams.
- probabilities: A dictionary containing probabilities.

Output:

- None (prints probabilities).

Steps: For each N-gram:

1. Extract the prefix (all elements except the last one) and the suffix (the last element).
2. Get the probability from the probabilities dictionary.
3. Print the prefix, suffix, and their associated probability.

Step-5: User Input

Prompt the user to enter a sentence. Prompt the user to enter the value of 'n' for N-grams (e.g., 1 for unigram, 2 for bigram, etc.).

Step-6: Generate N-grams and Calculate Probabilities

Call the generate_ngrams function to obtain a list of N-grams. Call the calculate_probability_matrix function to calculate probabilities.

Step-7: Print N-gram Probabilities

Call the print_ngram_probability function to display the N-grams and their associated probabilities.

Step-8: End of Code.

Flowchart:

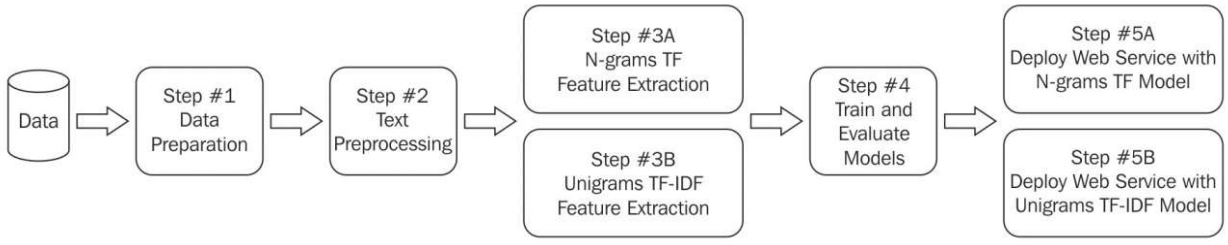


Fig. 3.3 - Flowchart of N-gram Program

Lab Outcomes:

Construct N-gram from a given corpus and calculate the probability of a sentence, as well as identify and classify named entities in a text into predefined categories.

Conclusions:

In conclusion, N-grams are a fundamental concept in NLP for capturing and modeling sequences of words or characters in text. They have various applications in language modeling, text analysis, and many other NLP tasks. Understanding the appropriate choice of N and employing techniques to address sparsity can significantly improve their effectiveness in NLP applications.

Program:

```
# Step-1
from collections import defaultdict
import random
# Step-2
def generate_ngrams(text, n):
words = text.split()
ngrams = [tuple(words[i:i+n]) for i in range(len(words)-n+1)]
return ngrams
# Step-3
def calculate_probability_matrix(ngrams):
counts = defaultdict(lambda: defaultdict(int))
for ngram in ngrams:
prefix = ngram[:-1]
suffix = ngram[-1]
counts[prefix][suffix] += 1
probabilities = {}
for prefix, suffix_counts in counts.items():
total_count = sum(suffix_counts.values())
```

```
probabilities[prefix] = suffix: count / total_count for suffix, count in suffix_counts.items()

return probabilities

# Step-4

def print_ngram_probability(ngrams, probabilities):

for ngram in ngrams:

prefix = ngram[:-1]

suffix = ngram[-1]

probability = probabilities[prefix][suffix]

print(f"prefix -> suffix: Probability = probability:.4f")

text = input("Enter a sentence: ")

# Step-5

n = int(input("Enter the value of 'n' for n-grams (1 for unigram, 2 for bigram, 3 for trigram, etc.): "))

# Step-6

ngrams = generate_ngrams(text, n)

probabilities = calculate_probability_matrix(ngrams)


# Step-7

print_ngram_probability(ngrams, probabilities)
```

Program Output Screenshot:

```
Enter a sentence: I went to the store, but they were closed, so I had to go to another store.
Enter the value of 'n' for n-grams (1 for unigram, 2 for bigram, 3 for trigram, etc.): 2
('I',) -> went: Probability = 0.5000
('went',) -> to: Probability = 1.0000
('to',) -> the: Probability = 0.3333
('the',) -> store,: Probability = 1.0000
('store,',) -> but: Probability = 1.0000
('but',) -> they: Probability = 1.0000
('they',) -> were: Probability = 1.0000
('were',) -> closed,: Probability = 1.0000
('closed,',) -> so: Probability = 1.0000
('so',) -> I: Probability = 1.0000
('I',) -> had: Probability = 0.5000
('had',) -> to: Probability = 1.0000
('to',) -> go: Probability = 0.3333
('go',) -> to: Probability = 1.0000
('to',) -> another: Probability = 0.3333
('another',) -> store.: Probability = 1.0000
```

nlp-lab3-1

September 2, 2024

1 Exprement NO: 3

1.0.1 objective: To Calculate thw probability of Sentence

1) Unigram
2) Bigram
3) Trigram]

[22]:
```
import nltk
from nltk.tokenize import sent_tokenize, word_tokenize
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
from nltk.stem import PorterStemmer
import matplotlib.pyplot as plt


# Download required NLTK resources
nltk.download('punkt')
nltk.download('stopwords')
nltk.download('wordnet')


def word_token(text):
    # Tokenization (word-level)
    words = word_tokenize(text)
    # Filtration (remove punctuation and convert to lowercase)
    filtered_words1 = [word.lower() for word in words if word.isalnum()]
    # Stop Word Removal
    stop_words = set(stopwords.words('english'))
    filtered_tokens2 = [t for t in filtered_words1 if t not in stop_words]
    # Stemming
    stemmer = PorterStemmer()
    stemmed_tokens = [stemmer.stem(t) for t in filtered_tokens2]
    # Lemmatization
    lemmatizer = WordNetLemmatizer()
    lemmatized_words = [lemmatizer.lemmatize(word) for word in stemmed_tokens]
    return␣
 ↪words,filtered_words1,filtered_tokens2,stemmed_tokens,lemmatized_words
```

```
[nltk_data] Downloading package punkt to /root/nltk_data…
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package stopwords to /root/nltk_data…
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package wordnet to /root/nltk_data…
[nltk_data]   Package wordnet is already up-to-date!
```

[23]:
```
# Take input
str1="In the scorching desert heat, I stumbled upon an oasis, an osis of water␣
 ↪that seemed like a mirage in the unforgiving wilderness"
# value=input()
sentence=value
sentence=str1
input=sentence
print(f"\nOriginal sentence :{sentence}\n")
```

```
Original sentence :In the scorching desert heat, I stumbled upon an oasis, an
osis of water that seemed like a mirage in the unforgiving wilderness
```

[24]:
```
# process the input and produce outcome of tokenization, filtering, stop word␣
 ↪removal, stemming, and lemmatization."""
output=word_token(sentence)
print(f"original sentence : {sentence}\nToken:\t\t\t {output[0]}")
```

```
original sentence : In the scorching desert heat, I stumbled upon an oasis, an
osis of water that seemed like a mirage in the unforgiving wilderness
Token:                  ['In', 'the', 'scorching', 'desert', 'heat', ',', 'I',
'stumbled', 'upon', 'an', 'oasis', ',', 'an', 'osis', 'of', 'water', 'that',
'seemed', 'like', 'a', 'mirage', 'in', 'the', 'unforgiving', 'wilderness']
```

#Unigram

[25]:
```
sentence=output[0]
dic=dict({})
for word in sentence:
  dic[word]=input.count(word)
```

[26]:
```
dic
```

[26]:
```
{'In': 1,
 'the': 2,
 'scorching': 1,
 'desert': 1,
 'heat': 1,
 ',': 2,
 'I': 2,
```

```
 'stumbled': 1,
 'upon': 1,
 'an': 2,
 'oasis': 1,
 'osis': 1,
 'of': 1,
 'water': 1,
 'that': 1,
 'seemed': 1,
 'like': 1,
 'a': 8,
 'mirage': 1,
 'in': 3,
 'unforgiving': 1,
 'wilderness': 1}
```

[27]:
```
probablity=dict({})
for key,value in dic.items():
  probablity[key]=value/len(sentence)
```

[28]:
```
print("Probablity : ")
for key,value in probablity.items():
  print(f"{key}\t : {value}")
probability_unigram=probablity
```

```
Probablity :
In       : 0.04
the      : 0.08
scorching        : 0.04
desert   : 0.04
heat     : 0.04
,        : 0.08
I        : 0.08
stumbled         : 0.04
upon     : 0.04
an       : 0.08
oasis    : 0.04
osis     : 0.04
of       : 0.04
water    : 0.04
that     : 0.04
seemed   : 0.04
like     : 0.04
a        : 0.32
mirage   : 0.04
in       : 0.12
unforgiving      : 0.04
```

```
wilderness       : 0.04
```

#Bigram

[29]:
```
sentence=[]
ranged=int(len(output[0])/2+1)
start=0
end=start+2
for x in range(ranged):
  tem=output[0][start:end]
  start=end
  end=end+2
  sentence.append(" ".join(tem))
```

[30]:
```
sentence
```

[30]:
```
['In the',
 'scorching desert',
 'heat ,',
 'I stumbled',
 'upon an',
 'oasis ,',
 'an osis',
 'of water',
 'that seemed',
 'like a',
 'mirage in',
 'the unforgiving',
 'wilderness']
```

[31]:
```
dic=dict({})
for word in sentence:
  dic[word]=input.count(word)
```

[32]:
```
dic
```

[32]:
```
{'In the': 1,
 'scorching desert': 1,
 'heat ,': 0,
 'I stumbled': 1,
 'upon an': 1,
 'oasis ,': 0,
 'an osis': 1,
 'of water': 1,
 'that seemed': 1,
 'like a': 1,
 'mirage in': 1,
 'the unforgiving': 1,
```

```
 'wilderness': 1}
```

[33]:
```
probablity=dict({})
for key,value in dic.items():
  probablity[key]=value/len(sentence)
```

[34]:
```
print("Probablity : ")
for key,value in probablity.items():
  print(f"{key}\t : {value}")
probability_bigram=probablity
```

```
Probablity :
In the    : 0.07692307692307693
scorching desert          : 0.07692307692307693
heat ,    : 0.0
I stumbled        : 0.07692307692307693
upon an   : 0.07692307692307693
oasis ,   : 0.0
an osis   : 0.07692307692307693
of water          : 0.07692307692307693
that seemed       : 0.07692307692307693
like a    : 0.07692307692307693
mirage in         : 0.07692307692307693
the unforgiving   : 0.07692307692307693
wilderness        : 0.07692307692307693
```

#Trigram

[35]:
```
sentence=[]
ranged=int(len(output[0])/3)
start=0
end=start+3
for x in range(ranged):
  tem=output[0][start:end]
  start=end
  end=end+3
  sentence.append(" ".join(tem))
```

[36]:
```
sentence
```

[36]:
```
['In the scorching',
 'desert heat ,',
 'I stumbled upon',
 'an oasis ,',
 'an osis of',
 'water that seemed',
 'like a mirage',
 'in the unforgiving']
```

[37]:
```
dic=dict({})
for word in sentence:
  dic[word]=input.count(word)
```

[38]:
```
dic
```

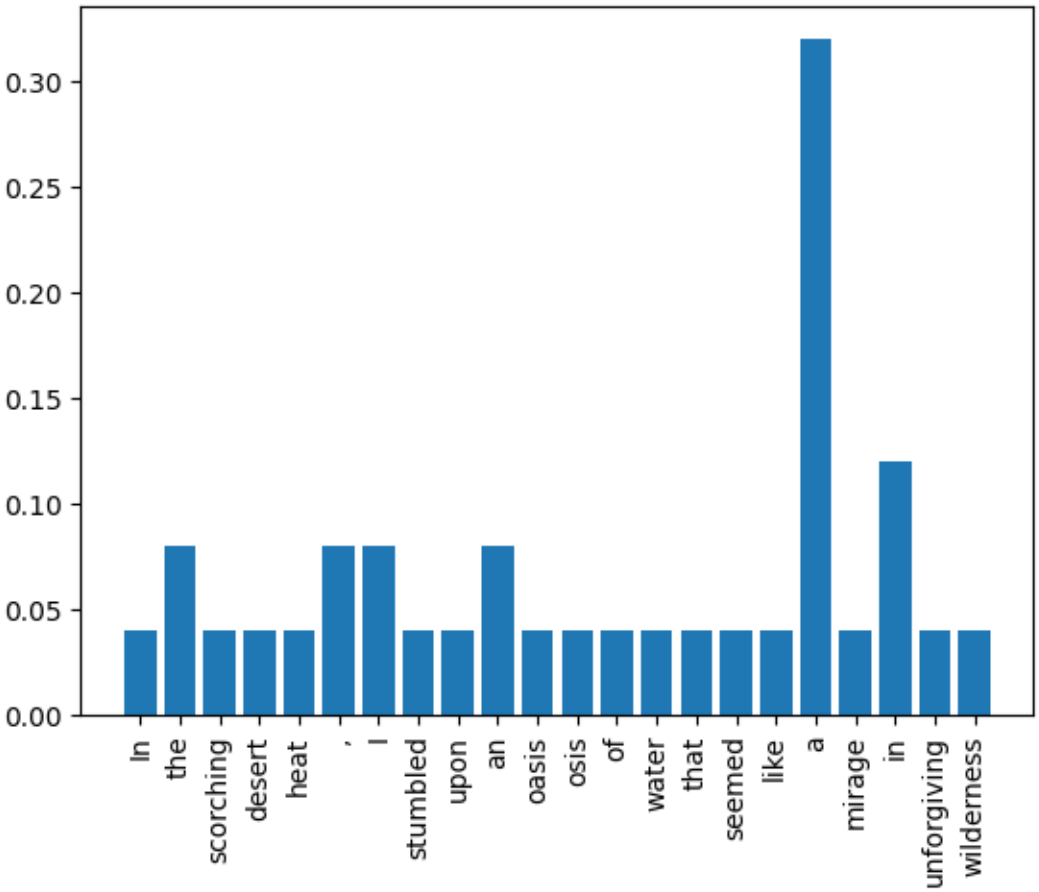
[38]:
```
{'In the scorching': 1,
 'desert heat ,': 0,
 'I stumbled upon': 1,
 'an oasis ,': 0,
 'an osis of': 1,
 'water that seemed': 1,
 'like a mirage': 1,
 'in the unforgiving': 1}
```

[39]:
```
probablity=dict({})
for key,value in dic.items():
  probablity[key]=value/len(sentence)
probability_trigram=probablity
```

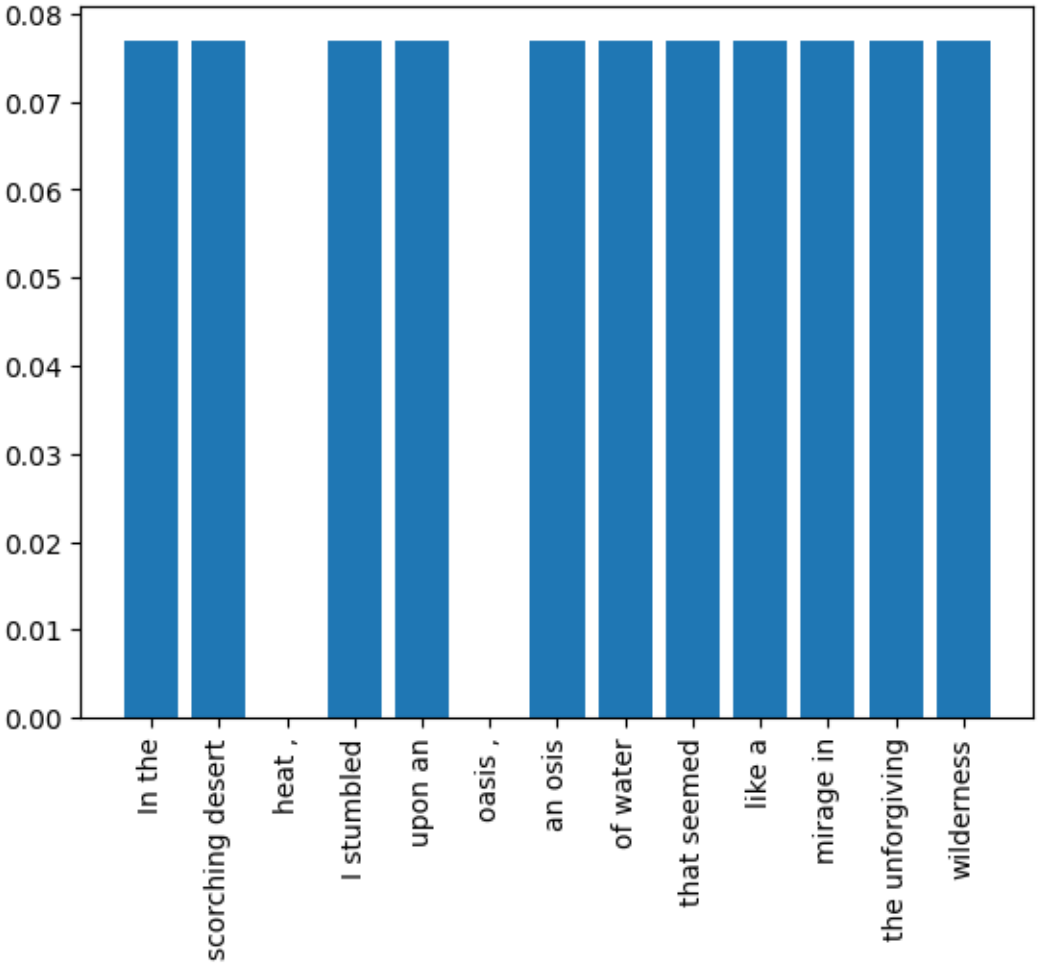
[47]:
```
print("Probablity : ")
for key,value in probability_unigram.items():
  print(f"{key}\t : {value}")

plt.bar(probability_unigram.keys(),probability_unigram.values())
plt.xticks(rotation='vertical')
plt.show()
print("Probablity : ")
for key,value in probability_bigram.items():
  print(f"{key}\t : {value}")

plt.bar(probability_bigram.keys(),probability_bigram.values())
plt.xticks(rotation='vertical')
plt.show()
probaility_trigram=probablity
print("Probablity : ")
for key,value in probability_trigram.items():
  print(f"{key}\t : {value}")

plt.bar(probability_trigram.keys(),probability_trigram.values())
plt.xticks(rotation='vertical')
plt.show()
```

```
Probablity :
In       : 0.04
the      : 0.08
scorching        : 0.04
```

```
desert    : 0.04
heat      : 0.04
,         : 0.08
I         : 0.08
stumbled          : 0.04
upon      : 0.04
an        : 0.08
oasis     : 0.04
osis      : 0.04
of        : 0.04
water     : 0.04
that      : 0.04
seemed    : 0.04
like      : 0.04
a         : 0.32
mirage    : 0.04
in        : 0.12
unforgiving       : 0.04
wilderness        : 0.04
```

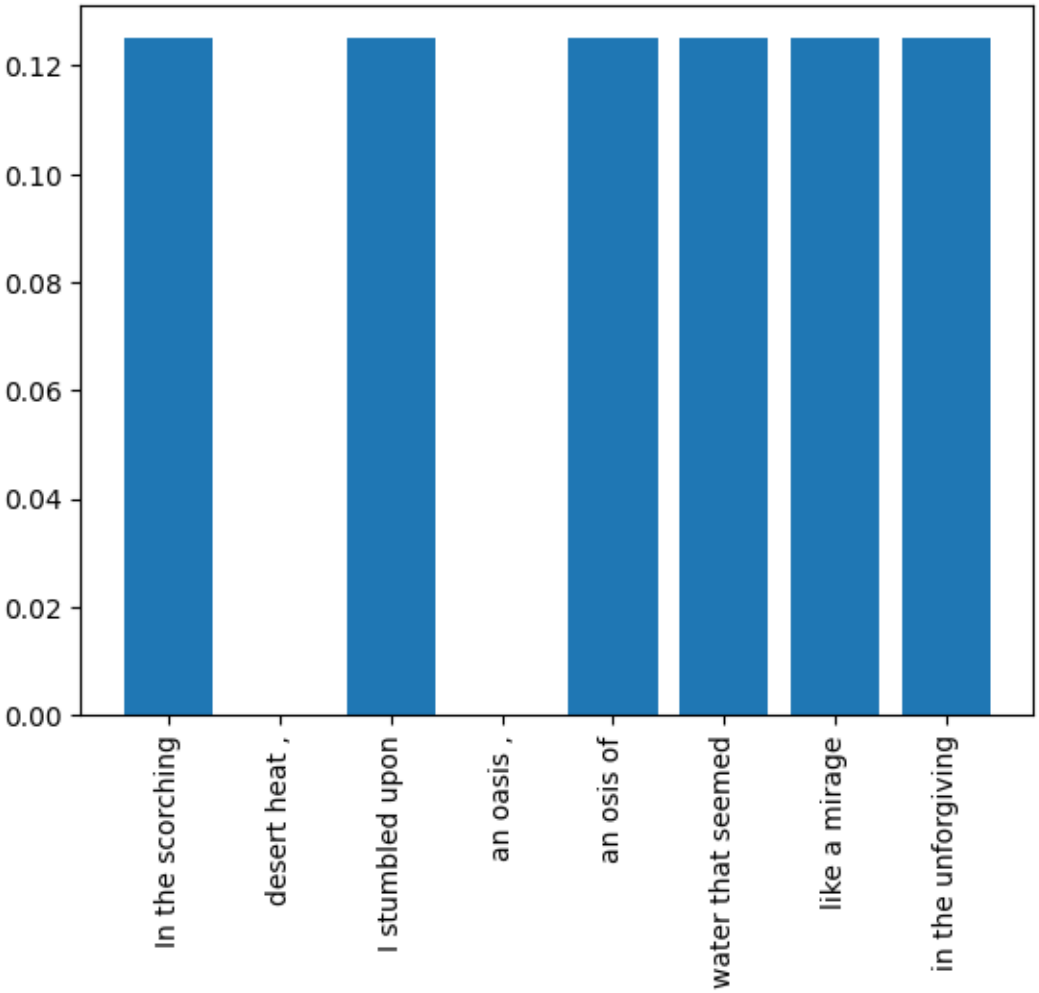


```
Probablity :
In the    : 0.07692307692307693
scorching desert          : 0.07692307692307693
heat ,    : 0.0
I stumbled        : 0.07692307692307693
upon an   : 0.07692307692307693
oasis ,   : 0.0
an osis   : 0.07692307692307693
of water          : 0.07692307692307693
that seemed       : 0.07692307692307693
like a    : 0.07692307692307693
mirage in         : 0.07692307692307693
the unforgiving   : 0.07692307692307693
wilderness        : 0.07692307692307693
```




```
Probablity :
In the scorching          : 0.125
```

```
desert heat ,     : 0.0
I stumbled upon   : 0.125
an oasis ,        : 0.0
an osis of        : 0.125
water that seemed         : 0.125
like a mirage     : 0.125
in the unforgiving        : 0.125
```



[40]:

Experiment No. 4

Title:

POS Tagging Chunking.

Aim:

The aim of this practical is to implement and understand POS tagging and chunking techniques for natural language processing tasks.

Lab Objectives:

Apply knowledge of parts of speech, understand the concept of chunking, and get familiar with the basic chunk tagset.

Theory:

Part-of-Speech (POS) Tagging:

Part-of-speech tagging, often abbreviated as POS tagging, is a fundamental task in natural language processing. It involves assigning grammatical categories (such as nouns, verbs, adjectives, etc.) to every word in a given text. This is crucial for understanding the syntactic structure of a sentence and extracting meaning from it.

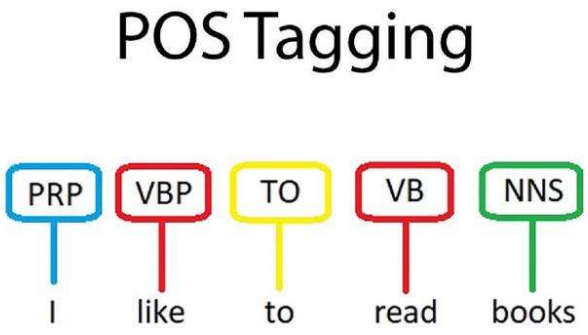


Fig. 4.1 - Common POS Tag

Details of POS tagging:

What is POS Tagging?

POS tagging is the process of automatically assigning a grammatical category (part of speech) to each word in a sentence. In the sentence "He runs quickly", POS tagging would label "He" as a pronoun, "runs" as a verb, and "quickly" as an adverb.

Importance of POS Tagging:

- Syntactic Analysis: POS tagging helps in understanding the grammatical structure of a sentence, which is crucial for more advanced NLP tasks like parsing.
- Disambiguation: It helps resolve ambiguities that arise from homonyms (words with the same spelling but different meanings) and polysemy (words with multiple meanings).
- Information Extraction: POS tags can be used to identify entities, and relationships, and extract useful information from text.

Common POS Tags:

- Noun (NN): Represents a person, place, thing, or idea.
- Verb (VB): Describes an action or state of being.
- Adjective (JJ): Modifies or describes a noun.
- Adverb (RB): Modifies or describes a verb, adjective, or other adverbs.
- Pronoun (PRP): Takes the place of a noun (e.g., he, she, it).
- Preposition (IN): Shows a relationship between a noun/pronoun and other words in a sentence.
- Conjunction (CC): Joins words, phrases, or clauses together.
- Determiner (DT): Determines or limits a noun.
- Interjection (UH): Expresses strong feelings or emotions.

Chunking: Chunking is the process of grouping words together based on their POS tags. These groups of words are called" chunks. "Chunking is used to identify phrases and extract meaningful information from text. It allows for a higher level of syntactic analysis beyond individual words. In the sentence "He runs quickly", chunking might identify the chunk "He" as a noun phrase and "quickly" as an adverb phrase.

Common Chunk Types:

- Noun Phrase (NP): A group of words centered around a noun.
- Verb Phrase (VP): A group of words centered around a verb.
- Prepositional Phrase (PP): A group of words starting with a preposition.
- Adjective Phrase (ADJP): A group of words centered around an adjective.
- Adverb Phrase (ADVP): A group of words centered around an adverb.
- Clausal Phrase (CP): A group of words centered around a clause.

Applications: Information Extraction: Chunking is used to extract specific types of information from text, such as names of people, locations, or organizations.

Parsing: Chunking forms the basis for more complex syntactic analysis, including parsing, which aims to understand the grammatical structure of sentences.

Challenges:

- Ambiguity: Like with POS tagging, chunking may face challenges with ambiguous sentences where multiple interpretations are possible.
- Variability: The structure of language can vary widely across different domains, making it challenging to create a one-size-fits-all chunking model.

Examples:

Consider the sentence: "The black cat chased the white mouse."

Here's the sentence with POS tags:

"The" -> Determiner (DT)

"black" -> Adjective (JJ)

"cat" -> Noun (NN)

"chased" -> Verb (VBD)

"the" -> Determiner (DT)

"white" -> Adjective (JJ)

"mouse" -> Noun (NN)

Chunking Example:

Noun Phrases (NP): "The black cat", "the white mouse"

Verb Phrases (VP): "chased"

Prepositional Phrases (PP): None in this sentence.

Algorithm:

Step-1: Import the necessary modules from the NLTK library.

Step-2: Download the required data for Named Entity Chunking using the nltk.download() function.

Step-3: Define a text string that you want to process.

Step-4: Tokenize the input text using the word_tokenize() function from NLTK. This converts the text into a list of words (tokens).

Step-5: Perform Part-of-Speech tagging on the tokens using pos_tag(). This assigns a grammatical category (noun, verb, etc.) to each word.

Step-6: Apply Named Entity Chunking using ne_chunk(). This identifies and groups words into named entities like organizations, locations, and persons.

Step-7: Print the POS-tagged text and the chunked text with named entities.

Flowchart:

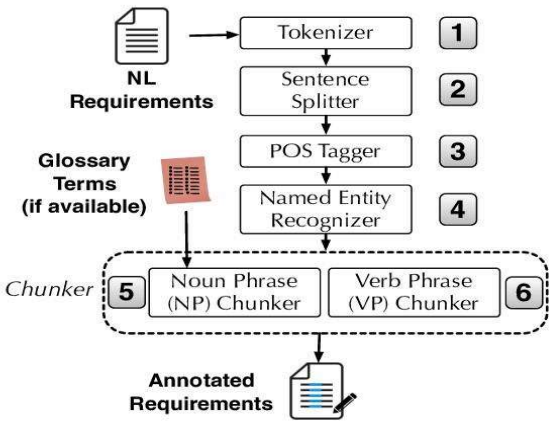


Fig. 4.2 - Flowchart of POS Tag chunking Program

Lab Outcomes:

Apply knowledge of parts of speech, understand the concept of chunking, and get familiar with the basic chunk tagset.

Conclusions:

In conclusion, the practical application of Part-of-Speech (POS) tagging and chunking is crucial in natural language processing (NLP) and information extraction. These techniques play a pivotal role by accurately identifying and categorizing the grammatical components of a text, enabling machines to comprehend human language more effectively. POS tagging assigns linguistic labels to words, facilitating tasks like sentiment analysis. Chunking groups words into meaningful chunks, aiding in the extraction of relevant information and identification of syntactic structures. Together, POS tagging and chunking enhance NLP systems, contributing to advancements in language understanding and machine learning applications.

Program:

```
# Step-1

import nltk

from nltk import word_tokenize, pos_tag, ne_chunk

# Step-2

nltk.download('maxent_ne_chunker')

nltk.download('words')

# Step-3

text = "Apple Inc. is a technology company based in Cupertino, California."

# Step-4

tokens = word_tokenize(text)

# Step-5

pos_tags = pos_tag(tokens)

# Step-6

chunked = ne_chunk(pos_tags)
```

```
# Step-7

print("POS tagged text:")

print(pos_tags)

print("text with named entities:")

print(chunked)
```

Program Output Screenshots:

```
[nltk_data] Downloading package maxent_ne_chunker to
[nltk_data]     C:\Users\Mahesh\AppData\Roaming\nltk_data...
[nltk_data]   Package maxent_ne_chunker is already up-to-date!
[nltk_data] Downloading package words to
[nltk_data]     C:\Users\Mahesh\AppData\Roaming\nltk_data...
POS tagged text:
[('Apple', 'NNP'), ('Inc.', 'NNP'), ('is', 'VBZ'), ('a', 'DT'), ('technology', 'NN'), ('company', 'NN'), ('based', 'VBN'), ('i
n', 'IN'), ('Cupertino', 'NNP'), (',', ','), ('California', 'NNP'), ('.', '.')]

Chunked text with named entities:
(S
  (PERSON Apple/NNP)
  (ORGANIZATION Inc./NNP)
  is/VBZ
  a/DT
  technology/NN
  company/NN
  based/VBN
  in/IN
  (GPE Cupertino/NNP)
  ,/,
  (GPE California/NNP)
  ./.)
[nltk_data]   Unzipping corpora\words.zip.
```

Experiment No. 5

Title:

Name Entity Recognition

Aim:

The aim of the Name Entity Recognition practical is to develop and implement a model that accurately identifies and classifies named entities in text, such as names of people, organizations, and locations.

Lab Objectives:

To create based on the NLP concept.

Theory:

Named Entity Recognition (NER) is a crucial task in Natural Language Processing (NLP) that involves identifying and classifying named entities in a text into predefined categories. Named entities refer to specific pieces of information within a text, such as names of people, locations, organizations, dates, and more. The primary goal of NER is to extract and categorize these entities accurately.

Here's a detailed explanation of NER:

1. **Tokenization:** The first step in NER is tokenization, where the input text is divided into individual tokens, usually words or subworlds. For example, the sentence" Barack Obama was born in Hawaii." would be tokenized into ["Barack", "Obama", "was", "born", "in", "Hawaii", "."].
2. **Part-of-Speech (POS) Tagging:** Before performing NER, it's often beneficial to perform Part-of-Speech tagging. This assigns a grammatical label (e.g., noun, verb, adjective) to each token. For example, "Barack" and "Obama" would be tagged as proper nouns.
3. **NER Labeling:** This is where the actual NER task happens. Each token is labeled with its corresponding entity category.

Common categories include:

- PERSON: Names of people.
- LOCATION: Names of places or locations.
- ORGANIZATION: Names of companies, institutions, etc.
- DATE: Any sort of date or time reference.

- TIME: Specific times mentioned.
- MONEY: Monetary values.
- PERCENT: Percentage values. - MISC: Miscellaneous entities.

4. **NER Techniques:**

- **Rule-Based Systems:** These rely on predefined rules and patterns to identify named entities. For instance, capitalization patterns, surrounding words, and POS tags can be used.
- **Machine Learning Models:** Techniques like Conditional Random Fields (CRF), Support Vector Machines (SVM), and deep learning models like Bidirectional LSTMs or Transformers can be used for NER.
- **Pretrained Models:** Models like BERT, GPT, etc., which are pretrained on massive corpora, can be fine-tuned for NER tasks.

5. **Evaluation:** NER models are evaluated using metrics like Precision, Recall, and F1-score, which measure the accuracy of entity recognition.
6. **Applications:**

NER has a wide range of applications, including:

- Information retrieval
- Question answering systems
- Machine translation
- Sentiment analysis
- Named Entity Linking (associating entities with knowledge base entries)

Examples:

Let's consider a more complex example: "Apple is headquartered in Cupertino, California. Tim Cook is the CEO of Apple."

- **Tokenized:** ["Apple", "is", "headquartered", "in", "Cupertino", ",", "California", ".", "Tim", "Cook", "is", "the", "CEO", "of", "Apple", "."]
- **POS Tagged:** [(Apple, NOUN), (is, VERB), (headquartered, VERB), (in, ADP), (Cupertino, NOUN), (,, PUNCT), (California, NOUN), (., PUNCT), (Tim, NOUN), (Cook, NOUN), (is, VERB), (the, DET), (CEO, NOUN), (of, ADP), (Apple, NOUN), (., PUNCT)]
- **NER Labeled:** [(Apple, ORGANIZATION), (Cupertino, LOCATION), (California, LOCATION), (Tim Cook, PERSON), (Apple, ORGANIZATION)]

Algorithm:

Step-1: Import the spaCy library.

Step-2: Load the English language model (en_core_web_sm) using spacy.load() method.

Step-3: Define a text string that you want to process.

Step-4: Initialize the spaCy model with the provided text using nlp(text). This creates a Doc object that contains linguistic annotations.

Step-5: Iterate through the entities identified in the document using a list comprehension, extracting both the entity text and its label.

Step-6: Print the identified named entities.

Flowchart:

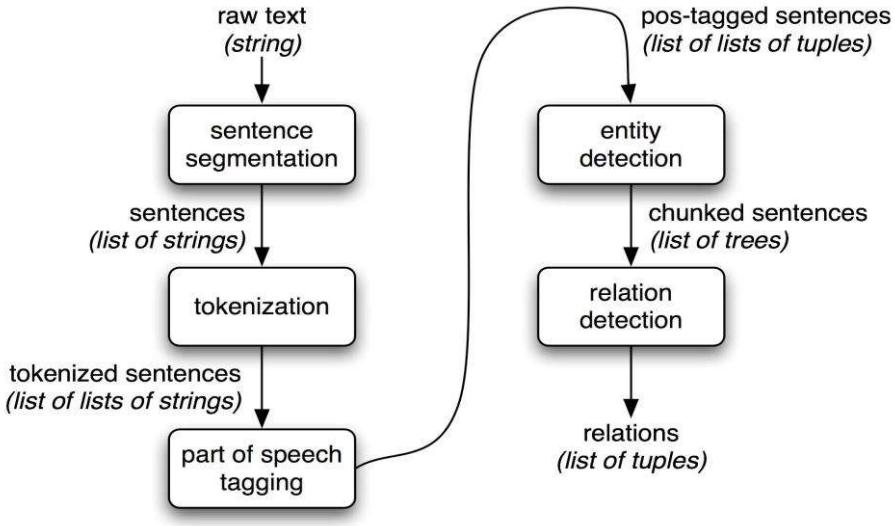


Fig. 5.1 - Flowchart of NER Program

Lab Outcomes:

Create based in the NLP concept.

Conclusions:

In summary, Named Entity Recognition (NER) is a fundamental aspect of natural language processing, facilitating the identification and classification of entities like people, organizations, locations, and dates in a text. Employing machine learning algorithms, NER is essential in practical applications such as information extraction, sentiment analysis, and question-answering systems. Its capacity to discern and categorize entities enhances language understanding, automates tasks involving unstructured text data, and significantly contributes to developing intelligent and

context-aware systems.

Program:

```
# Step-1

import spacy

# Step-2

nlp = spacy.load(”en_core_web_sm”)

# Step-3

text = ”Barack Obama was the 44th President of the United States. He was born in Hawaii.”

# Step-4

doc = nlp(text)

# Step-5

entities = [(ent.text, ent.label_) for ent in doc.ents]

# Step-6

print(”Named Entities:”)

print(entities)
```

Output:

```
Named Entities:
[('Barack Obama', 'PERSON'), ('44th', 'ORDINAL'), ('the United States', 'GPE'), ('Hawaii', 'GPE')]
```