**Ex.no : 01 Word Analysis in NLP**

**Program**

import nltk

from nltk.corpus import wordnet

# Download required NLTK data

nltk.download('wordnet')

nltk.download('omw-1.4')

def analyze\_word(word):

synsets = wordnet.synsets(word)

if not synsets:

return "No analysis found."

result = f"\nAnalysis of '{word}':\n"

for syn in synsets[:3]:

synonyms = {lemma.name().replace('\_', ' ') for lemma in syn.lemmas()}

antonyms = {lemma.antonyms()[0].name().replace('\_', ' ') for lemma in syn.lemmas() if lemma.antonyms()}

result += f"\n- Definition: {syn.definition()}\n"

result += f"- Part of Speech: {syn.pos()}\n"

if syn.examples():

result += f"- Example: {syn.examples()[0]}\n"

result += f"- Synonyms: {', '.join(synonyms) or 'None'}\n"

result += f"- Antonyms: {', '.join(antonyms) or 'None'}\n"

result += "-" \* 30 + "\n"

return result

if \_\_name\_\_ == "\_\_main\_\_":

print(analyze\_word(input("Enter a word: ")))

**Output**

Enter a word: animal

Analysis of 'animal':

- Definition: a living organism characterized by voluntary movement

- Part of Speech: n

- Synonyms: beast, fauna, animate being, brute, animal, creature

- Antonyms: None

------------------------------

- Definition: marked by the appetites and passions of the body

- Part of Speech: s

- Example: animal instincts

- Synonyms: sensual, fleshly, animal, carnal

- Antonyms: None

**Ex.no: 02 Word Generation in NLP**

**Program**

import nltk

from nltk.corpus import wordnet as wn

# Download WordNet if not already downloaded

nltk.download('wordnet')

nltk.download('omw-1.4')

def get\_related\_words(word):

"""Fetch related words using WordNet synsets."""

synsets = wn.synsets(word)

related\_words = set()

for syn in synsets:

# Add synonyms

for lemma in syn.lemmas():

related\_words.add(lemma.name().replace("\_", " "))

# Add hypernyms and hyponyms for richer results

for hyper in syn.hypernyms():

for lemma in hyper.lemmas():

related\_words.add(lemma.name().replace("\_", " "))

for hypo in syn.hyponyms():

for lemma in hypo.lemmas():

related\_words.add(lemma.name().replace("\_", " "))

return list(related\_words)[:10] # Limit to 10 results

if \_\_name\_\_ == "\_\_main\_\_":

print("WordNet-based Word Explorer")

word = input("Enter a word: ").lower().strip()

related = get\_related\_words(word)

if related:

print("\nRelated Words:", ", ".join(related))

else:

print("\nNo related words found in WordNet.")

**Output:**

Enter a word: animal

Related Words: purebred, adult, carnal, herbivore, sea animal, migrator, brute, omnivore, domesticated animal, molter

**Ex.no: 03 Morphological Analysis in NLP**

**Program**

import nltk

from nltk.stem import PorterStemmer, WordNetLemmatizer

# Download necessary NLTK resources

nltk.download('wordnet')

nltk.download('omw-1.4')

def morphology\_analysis(word):

# Create instances for stemmer and lemmatizer

ps = PorterStemmer()

lemmatizer = WordNetLemmatizer()

# Perform stemming and lemmatization

stemmed\_word = ps.stem(word)

lemmatized\_word = lemmatizer.lemmatize(word)

# Display the results

print("\nOriginal Word:", word)

print("Stemmed Word:", stemmed\_word)

print("Lemmatized Word:", lemmatized\_word)

# Main function

if \_\_name\_\_ == "\_\_main\_\_":

word = input("Enter a word for morphological analysis: ")

morphology\_analysis(word)

**Output:**

Enter a word for morphological analysis: nature

Original Word: nature

Stemmed Word: natur

Lemmatized Word: nature

**Ex.no: 04 N-Gram Analysis in NLP**

**Program**

import re

from collections import defaultdict

def preprocess(text):

"""Convert text to lowercase and remove punctuation."""

return re.sub(r'[^\w\s]', '', text.lower())

def generate\_ngrams(text, n):

"""Generate n-grams from text."""

words = text.split()

return [tuple(words[i:i+n]) for i in range(len(words) - n + 1)]

def ngram\_model(text, n):

"""Create an n-gram model with frequency counts."""

ngrams = generate\_ngrams(preprocess(text), n)

ngram\_count = defaultdict(int)

for ngram in ngrams:

ngram\_count[ngram] += 1

return ngram\_count

# ---- User Interaction ----

print("N-gram Frequency Model")

user\_text = input("Enter your text:\n")

try:

n = int(input("\nEnter the n-gram size (e.g., 1 for unigrams, 2 for bigrams): "))

if n < 1:

raise ValueError

except ValueError:

print("Invalid n-gram size. Please enter a positive integer.")

else:

model = ngram\_model(user\_text, n)

print(f"\n{n}-gram Frequencies:")

for k, v in model.items():

print(f"{k}: {v}")

**Output**

N-gram Frequency Model

3-gram Frequencies:

('the', 'quick', 'brown'): 1

('quick', 'brown', 'fox'): 1

('brown', 'fox', 'jumps'): 1

('fox', 'jumps', 'over'): 1

('jumps', 'over', 'the'): 1

('over', 'the', 'lazy'): 1

('the', 'lazy', 'dog'): 1

**Ex.no:05 N-Gram Smoothing in NLP**

**Program**

import nltk

from nltk.util import ngrams

from collections import defaultdict, Counter

# Download required resources

nltk.download('punkt')

def generate\_ngrams(text, n):

words = nltk.word\_tokenize(text.lower()) # Tokenization

ngram\_list = list(ngrams(words, n)) # Generate n-grams

return ngram\_list

def compute\_smoothed\_probabilities(ngrams\_list, ngram\_counts, unigram\_counts, vocab\_size):

smoothed\_probs = {}

for ngram in ngrams\_list:

prefix = ngram[:-1] # Previous (n-1) words

word = ngram[-1] # Current word

count\_ngram = ngram\_counts[ngram]

count\_prefix = unigram\_counts[prefix] if prefix in unigram\_counts else 0

# Apply Laplace Smoothing

smoothed\_prob = (count\_ngram + 1) / (count\_prefix + vocab\_size)

smoothed\_probs[ngram] = smoothed\_prob

return smoothed\_probs

# Main function

def main():

text = input("Enter a sentence: ")

n = int(input("Enter the value of N for N-grams: "))

# Generate n-grams

ngrams\_list = generate\_ngrams(text, n)

# Compute n-gram and (n-1)-gram counts

ngram\_counts = Counter(ngrams\_list)

if n > 1:

prefix\_ngrams = generate\_ngrams(text, n - 1)

unigram\_counts = Counter(prefix\_ngrams)

else:

unigram\_counts = Counter(nltk.word\_tokenize(text.lower()))

# Calculate vocabulary size

vocab\_size = len(set(nltk.word\_tokenize(text.lower())))

# Compute smoothed probabilities

smoothed\_probs = compute\_smoothed\_probabilities(ngrams\_list, ngram\_counts, unigram\_counts, vocab\_size)

# Display results

print("\nN-grams and their Smoothed Probabilities:")

for ngram, prob in smoothed\_probs.items():

print(f"{ngram} -> {prob:.4f}")

if \_\_name\_\_ == "\_\_main\_\_":

main()

**Output**

Enter a sentence: the dog chases the cat

Enter the value of N for N-grams: 2

N-grams and their Smoothed Probabilities:

('the', 'dog') -> 0.3333

('dog', 'chases') -> 0.4000

('chases', 'the') -> 0.4000

('the', 'cat') -> 0.3333

**Ex.no: 06 POS Tagging using Hidden Markov Model (HMM) in NLP**

**Program**

import nltk

from nltk.corpus import treebank

from nltk.tag import hmm

from nltk.tokenize import word\_tokenize

# Download required resources

nltk.download('treebank')

nltk.download('punkt')

# Load the Treebank corpus and split into training and test sets

train\_sents = treebank.tagged\_sents()[:3000]

test\_sents = treebank.tagged\_sents()[3000:]

# Initialize and train the HMM tagger

trainer = hmm.HiddenMarkovModelTrainer()

tagger = trainer.train(train\_sents)

# Function to tag a new sentence

def pos\_tag\_sentence(sentence):

tokens = word\_tokenize(sentence)

tagged = tagger.tag(tokens)

return tagged

# Get user input

sentence = input("Enter a sentence to POS tag: ")

# Tag the sentence and display the result

tagged\_sentence = pos\_tag\_sentence(sentence)

print("\nTagged Sentence:")

for word, tag in tagged\_sentence:

print(f"{word}: {tag}")

**Output**

Enter a sentence to POS tag: She enjoys reading books

Tagged Sentence:

She: PRP

enjoys: NNP

reading: NNP

books: NNP

**Ex.no: 07 POS Tagging using Viterbi Decoding**

**Program**

import numpy as np

# POS tags (states) and probabilities

states = ['Noun', 'Verb', 'Adjective']

start\_prob = {'Noun': 0.5, 'Verb': 0.3, 'Adjective': 0.2}

transition\_prob = {

'Noun': {'Noun': 0.1, 'Verb': 0.6, 'Adjective': 0.3},

'Verb': {'Noun': 0.4, 'Verb': 0.3, 'Adjective': 0.3},

'Adjective': {'Noun': 0.5, 'Verb': 0.2, 'Adjective': 0.3}

}

emission\_prob = {

'Noun': {'dog': 0.4, 'cat': 0.4, 'runs': 0.1, 'fast': 0.1},

'Verb': {'dog': 0.1, 'cat': 0.1, 'runs': 0.6, 'fast': 0.2},

'Adjective': {'dog': 0.1, 'cat': 0.1, 'runs': 0.2, 'fast': 0.6}

}

def viterbi(sentence):

words = sentence.lower().split()

n, m = len(words), len(states)

viterbi\_matrix = np.zeros((m, n))

backpointer = np.zeros((m, n), dtype=int)

# Initialization

for i, state in enumerate(states):

viterbi\_matrix[i, 0] = start\_prob[state] \* emission\_prob[state].get(words[0], 0.01)

# Recursion

for t in range(1, n):

for i, state in enumerate(states):

max\_prob, max\_state = max(

[(viterbi\_matrix[j, t-1] \* transition\_prob[states[j]][state] \* emission\_prob[state].get(words[t], 0.01), j)

for j in range(m)],

key=lambda x: x[0]

)

viterbi\_matrix[i, t] = max\_prob

backpointer[i, t] = max\_state

# Termination

best\_last\_state = np.argmax(viterbi\_matrix[:, -1])

best\_path = [best\_last\_state]

# Backtrace

for t in range(n - 1, 0, -1):

best\_path.insert(0, backpointer[best\_path[0], t])

tagged\_sequence = [(words[i], states[state]) for i, state in enumerate(best\_path)]

return tagged\_sequence

# User interaction

sentence = input("Enter a sentence: ")

print("\nPOS Tagged Sentence:")

for word, tag in viterbi(sentence):

print(f"{word} ---> {tag}")

**Output:**

Enter a sentence: apple is a fruit

POS Tagged Sentence:

apple ---> Noun

is ---> Verb

a ---> Noun

fruit ---> Verb

**Ex.no: 08 Building a POS Tagger in NLP**

**Program**

import nltk

import ssl

try:

# Bypass SSL verification (only if SSL errors occur)

try:

\_create\_unverified\_https\_context = ssl.\_create\_unverified\_context

except AttributeError:

pass

else:

ssl.\_create\_default\_https\_context = \_create\_unverified\_https\_context

# Correct resource download

nltk.download('punkt')

nltk.download('averaged\_perceptron\_tagger')

def pos\_tagger(sentence):

words = nltk.word\_tokenize(sentence)

pos\_tags = nltk.pos\_tag(words)

print("\nPOS Tagging Results:")

for word, tag in pos\_tags:

print(f"{word} ---> {tag}")

sentence = input("Enter a sentence: ")

pos\_tagger(sentence)

except LookupError as e:

print(f"Error: Required NLTK data not found. Please run the following:\n\n"

"import nltk\nnltk.download('averaged\_perceptron\_tagger')\n\nError details: {e}")

except Exception as e:

print(f"An unexpected error occurred: {e}")

**Output**

Enter a sentence: dog runs fast than a cat

POS Tagging Results:

dog ---> NN

runs ---> VBZ

fast ---> RB

than ---> IN

a ---> DT

cat ---> NN

**Result**

Thus the program successfully builds a **POS tagger** and the output was verified.

**Ex.no: 09 Chunking in NLP**

**Program**

import nltk

from nltk.tokenize import word\_tokenize

from nltk.tag import pos\_tag

from nltk.chunk import RegexpParser

from nltk.tree import Tree

# Download necessary datasets

nltk.download('punkt')

nltk.download('averaged\_perceptron\_tagger')

# Input sentence

sentence = "The quick brown fox jumps over the lazy dog"

# Tokenize and POS tag

tokens = word\_tokenize(sentence)

tagged\_words = pos\_tag(tokens)

# Define chunk grammar (e.g., extracting noun phrases)

chunk\_grammar = r"""NP: {<DT>?<JJ>\*<NN|NNS>}""" # NP = Noun Phrase

# Create the chunk parser

chunk\_parser = RegexpParser(chunk\_grammar)

# Apply chunking

chunked\_output = chunk\_parser.parse(tagged\_words)

# Print chunked result

print("\nChunked Output:")

print(chunked\_output)

# Display chunked tree in text format

chunked\_output.pretty\_print()

**Output**

(S

(NP The/DT quick/JJ brown/JJ fox/NN)

jumps/VBZ

over/IN

(NP the/DT lazy/JJ dog/NN))

S

\_\_\_\_\_\_\_\_\_\_|\_\_\_\_\_\_\_\_\_\_

| | | NP

| | | \_\_\_\_\_|\_\_\_\_\_\_\_

NP | | DT JJ NN

| | | | | |

The jumps over the lazy dog

**Ex.no: 10 Building a Chunker in NLP**

**Program**

# Step 1: Install and import necessary libraries

import nltk

from nltk import word\_tokenize, pos\_tag

from nltk.chunk import RegexpParser

# Download required resources

nltk.download('punkt') # For tokenization

nltk.download('averaged\_perceptron\_tagger') # For POS tagging

# Step 2: Tokenize and perform POS tagging on a sample sentence

sentence = input("Enter the sentence: ")

tokens = word\_tokenize(sentence)

tagged\_tokens = pos\_tag(tokens)

print("POS Tagged Tokens:")

print(tagged\_tokens)

# Step 3: Define a custom chunking grammar using regular expressions

chunk\_grammar = r"""

NP: {<DT>?<JJ>\*<NN.\*>} # Noun Phrase

VP: {<VB.\*><NP|PP|CLAUSE>\*} # Verb Phrase

"""

# Step 4: Apply chunking using RegexpParser

chunk\_parser = RegexpParser(chunk\_grammar)

chunk\_tree = chunk\_parser.parse(tagged\_tokens)

# Step 5: Display and visualize the extracted chunks

print("\nChunked Sentence Tree:")

chunk\_tree.pprint()

# Optional: Visualize the chunk tree in a popup window (works only in desktop environments)

try:

chunk\_tree.draw()

except:

print("Tree visualization is not supported in this environment.")

**Output:**

enter the sting : "The quick brown fox jumps over the lazy dog"

POS Tagged Tokens:

[('``', '``'), ('The', 'DT'), ('quick', 'JJ'), ('brown', 'NN'), ('fox', 'NN'), ('jumps', 'VBZ'), ('over', 'IN'), ('the', 'DT'), ('lazy', 'JJ'), ('dog', 'NN'), ("''", "''")]

Chunked Sentence Tree:

(S

``/``

(NP The/DT quick/JJ brown/NN)

(NP fox/NN)

(VP jumps/VBZ)

over/IN

(NP the/DT lazy/JJ dog/NN)

''/'')