**Exp-1:** **Write program demonstrates how to use regular expressions in Python to match and search for patterns in text.**

**Code:**

import re

text = "Hello, my email is pranathi.sse@saveetha.com, and my phone number is 630-040-3626."

email\_pattern = r'\b[A-Za-z0-9.\_%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,7}\b'

phone\_pattern = r'\b\d{3}-\d{3}-\d{4}\b'

emails = re.findall(email\_pattern, text)

print("Email addresses found:")

for email in emails:

print(email)

phones = re.findall(phone\_pattern, text)

print("\nPhone numbers found:")

for phone in phones:

print(phone)

**Output:**

Email addresses found:

pranathi.sse@saveetha.com

Phone numbers found:

630-040-3626

**Exp-2:** **Implement a basic finite state automaton that recognizes a specific language or pattern. In this example, we’ll create a simple automaton to match strings ending with ‘ab’ using python.**

**Code:**

class StringMatcher:

def \_\_init\_\_(self):

self.current\_state = 0

def match(self, input\_string):

for char in input\_string:

if char == 'a':

self.current\_state = 1

elif char == 'b' and self.current\_state == 1:

self.current\_state = 2

else:

self.current\_state = 0

return self.current\_state == 2

matcher = StringMatcher()

input\_strings = ["ab", "abc", "aab", "abab", "cba"]

for input\_string in input\_strings:

if matcher.match(input\_string):

print(f"'{input\_string}' is accepted")

else:

print(f"'{input\_string}' is rejected")

**Output:**

'ab' is accepted

'abc' is rejected

'aab' is accepted

'abab' is accepted

'cba' is rejected

**Exp-3: Write program demonstrates how to perform morphological analysis using the NLTK library in Python.**

**Code:**

import nltk

from nltk import word\_tokenize

from nltk.corpus import wordnet

from nltk.stem import WordNetLemmatizer

text = "The quick brown foxes jumped over the lazy dogs."

tokens = word\_tokenize(text)

lemmatizer = WordNetLemmatizer()

lemmatized\_tokens = [lemmatizer.lemmatize(token, wordnet.VERB) for token in tokens]

print("Original Tokens:", tokens)

print("Lemmatized Tokens:", lemmatized\_tokens)

**Output:**

Original Tokens: ['The', 'quick', 'brown', 'foxes', 'jumped', 'over', 'the', 'lazy', 'dogs', '.']

Lemmatized Tokens: ['The', 'quick', 'brown', 'fox', 'jump', 'over', 'the', 'lazy', 'dog', '.']

**Exp-4:** Implement a finite-state machine for morphological parsing. In this example, we’ll create a simple machine to generate plural forms of English nouns using python.

**Code:**

class PluralStateMachine:

def \_\_init\_\_(self):

self.state = 'start'

def reset(self):

self.state = 'start'

def transition(self, input\_char):

transitions = {

'start': {

's': 'es',

'x': 'es',

'y': 'ies',

},

'start\_vowel': {

's': 's',

'x': 'x',

'y': 'ys',

},

}

if input\_char.isalpha():

input\_char = input\_char.lower()

if self.state in transitions and input\_char in transitions[self.state]:

plural\_suffix = transitions[self.state][input\_char]

return plural\_suffix

else:

return input\_char

def generate\_plural(self, noun):

plural = ''

self.reset()

for char in noun:

plural += self.transition(char)

return plural

noun = "Candy"

pluralizer = PluralStateMachine()

plural\_form = pluralizer.generate\_plural(noun)

print(f"The plural form of '{noun}' is '{plural\_form}'")

**Output:**

The plural form of 'Candy' is 'candies'

**Exp-5: Use the Porter Stemmer algorithm to perform word stemming on a list of words using python libraries.**

**Code:**

import nltk

from nltk.stem import PorterStemmer

stemmer = PorterStemmer()

words = ["jumping", "jumps", "jumped", "running", "flies", "flies", "stemming", "stemmer"]

stemmed\_words = [stemmer.stem(word) for word in words]

for i in range(len(words)):

print(f"{words[i]} -> {stemmed\_words[i]}")

**Output:**

jumping -> jump

jumps -> jump

jumped -> jump

running -> run

flies -> fli

flies -> fli

stemming -> stem

stemmer -> stemmer

**Exp-6: Implement a basic N-gram model for text generation. For example, generate text using a bigram model using python.**

**Code:**

import random

corpus = "This is a simple example of an n-gram model in Python. It generates text based on a given input text."

words = corpus.split()

bigrams = {}

for i in range(len(words) - 1):

current\_word = words[i]

next\_word = words[i + 1]

if current\_word in bigrams:

bigrams[current\_word].append(next\_word)

else:

bigrams[current\_word] = [next\_word]

def generate\_text(start\_word, num\_words):

generated\_text = [start\_word]

current\_word = start\_word

for \_ in range(num\_words - 1):

if current\_word in bigrams:

next\_word = random.choice(bigrams[current\_word])

generated\_text.append(next\_word)

current\_word = next\_word

else:

break

return " ".join(generated\_text)

start\_word = "This"

num\_words = 10

generated\_text = generate\_text(start\_word, num\_words)

print(generated\_text)

**Output:**

This is a given input text.

**Exp-7: Write program using the NLTK library to perform part-of-speech tagging on a text.**

**Code:**

import nltk

nltk.download('punkt')

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

text = "This is a sample sentence for part-of-speech tagging."

words = nltk.word\_tokenize(text)

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

print("Word\t\tPOS Tag")

print("-" \* 20)

for word, pos in pos\_tags:

print(f"{word}\t\t{pos}")

**Output:**

Word POS Tag

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

This DT

is VBZ

a DT

sample JJ

sentence NN

for IN

part-of-speech JJ

tagging NN

. .

**Exp-8:** **Implement a simple stochastic part-of-speech tagging algorithm using a basic probabilistic model to assign POS tags using python.**

**Code:**

import random

pos\_tags = ['Noun', 'Verb', 'Adjective', 'Adverb', 'Pronoun']

transition\_probabilities = {

'Start': {'Noun': 0.4, 'Verb': 0.3, 'Adjective': 0.1, 'Adverb': 0.1, 'Pronoun': 0.1},

'Noun': {'Noun': 0.2, 'Verb': 0.4, 'Adjective': 0.2, 'Adverb': 0.1, 'Pronoun': 0.1},

'Verb': {'Noun': 0.3, 'Verb': 0.2, 'Adjective': 0.2, 'Adverb': 0.2, 'Pronoun': 0.1},

'Adjective': {'Noun': 0.2, 'Verb': 0.3, 'Adjective': 0.2, 'Adverb': 0.2, 'Pronoun': 0.1},

'Adverb': {'Noun': 0.1, 'Verb': 0.2, 'Adjective': 0.2, 'Adverb': 0.3, 'Pronoun': 0.2},

'Pronoun': {'Noun': 0.2, 'Verb': 0.2, 'Adjective': 0.2, 'Adverb': 0.1, 'Pronoun': 0.3},

}

emission\_probabilities = {

'Noun': {'cat': 0.2, 'dog': 0.2, 'tree': 0.1, 'house': 0.1, 'man': 0.4},

'Verb': {'runs': 0.3, 'jumps': 0.2, 'sleeps': 0.2, 'eats': 0.2, 'walks': 0.1},

'Adjective': {'big': 0.2, 'small': 0.2, 'happy': 0.2, 'red': 0.2, 'green': 0.2},

'Adverb': {'quickly': 0.2, 'slowly': 0.2, 'happily': 0.2, 'loudly': 0.2, 'well': 0.2},

'Pronoun': {'he': 0.2, 'she': 0.2, 'it': 0.2, 'they': 0.2, 'we': 0.2},

}

def stochastic\_pos\_tagging(sentence):

tags = []

current\_state = 'Start'

words = sentence.split()

for word in words:

if word in emission\_probabilities['Noun']:

word\_pos = 'Noun'

elif word in emission\_probabilities['Verb']:

word\_pos = 'Verb'

elif word in emission\_probabilities['Adjective']:

word\_pos = 'Adjective'

elif word in emission\_probabilities['Adverb']:

word\_pos = 'Adverb'

elif word in emission\_probabilities['Pronoun']:

word\_pos = 'Pronoun'

else:

word\_pos = random.choice(pos\_tags)

tags.append(word\_pos)

next\_state = random.choices(pos\_tags, weights=list(transition\_probabilities[current\_state].values()))[0]

current\_state = next\_state

return list(zip(words, tags))

sentence = "The cat slowly walks happy loudly jumps"

tagged\_sentence = stochastic\_pos\_tagging(sentence)

print(tagged\_sentence)

**Output:**

[('The', 'Adjective'), ('cat', 'Noun'), ('slowly', 'Adverb'), ('walks', 'Verb'), ('happy', 'Adjective'), ('loudly', 'Adverb'), ('jumps', 'Verb')]

**Exp-9: Implement a rule-based part-of-speech tagging system using regular expressions using python.**

**Code:**

import re

patterns = [

(r'\b(?:\w\*ed|ing)\b', 'Verb'),

(r'\b(?:\w\*\'s)\b', 'Noun'),

(r'\b(?:The|A|An)\b', 'Determiner'),

(r'\b(?:\d+[\.,]?\d\*)\b', 'Number'),

(r'\b(?:\w+ly)\b', 'Adverb'),

(r'\b(?:\w+est)\b', 'Adjective'),

(r'\b(?:\w+er)\b', 'Adjective'),

(r'\b(?:\w+ing)\b', 'Verb'),

(r'\b(?:\w+ed)\b', 'Verb'),

(r'\b(?:\w+s)\b', 'Noun'),

(r'\b(?:\w+\'re)\b', 'Verb'),

(r'\b(?:\w+\'m)\b', 'Verb'),

(r'\b(?:\w+\'ve)\b', 'Verb'),

(r'\b(?:\w+\'ll)\b', 'Verb'),

(r'\b(?:\w+\'d)\b', 'Verb'),

(r'\b(?:\w+)\b', 'Noun'),

]

def rule\_based\_pos\_tagging(text):

tagged\_words = []

for word in text.split():

for pattern, tag in patterns:

if re.match(pattern, word, re.IGNORECASE):

tagged\_words.append((word, tag))

break

else:

tagged\_words.append((word, 'Noun'))

return tagged\_words

text = "I love running daily I run 2 kms as quickly as i can"

tagged\_text = rule\_based\_pos\_tagging(text)

for word, pos\_tag in tagged\_text:

print(f"{word}: {pos\_tag}")

**Output:**

I: Noun

love: Noun

running: Verb

daily: Adverb

I: Noun

run: Noun

2: Number

kms: Noun

as: Noun

quickly: Adverb

as: Noun

i: Noun

can: Noun

**Exp-10:** **Implement transformation-based tagging using a set of transformation rules, apply a simple rule to tag words using python.**

**Code:**

import re

transformation\_rules = [

(r'\b[Aa]pple\b', 'Noun'),

(r'\b\d+\b', 'Number'),

(r'\b\w+ly\b', 'Adverb'),

(r'\b\w+ing\b', 'Verb'),

(r'\b\w+ed\b', 'Verb'),

]

def transform\_based\_pos\_tagging(text):

tagged\_words = []

words = text.split()

for word in words:

tagged = False

for pattern, tag in transformation\_rules:

if re.match(pattern, word, re.IGNORECASE):

tagged\_words.append((word, tag))

tagged = True

break

if not tagged:

tagged\_words.append((word, 'Noun'))

return tagged\_words

text = "I ate an Apple and it tasted delicious."

tagged\_text = transform\_based\_pos\_tagging(text)

for word, pos\_tag in tagged\_text:

print(f"{word}: {pos\_tag}")

**Output:**

I: Noun

ate: Noun

an: Noun

Apple: Noun

and: Noun

it: Noun

tasted: Verb

delicious.: Noun

**Exp-11:** **Implement a simple top-down parser for context-free grammars using python.**

**Code:**

class ParseError(Exception):

pass

def parse\_expression(tokens):

term = parse\_term(tokens)

if tokens and tokens[0] == '+':

tokens.pop(0)

expression = parse\_expression(tokens)

return term + expression

return term

def parse\_term(tokens):

factor = parse\_factor(tokens)

if tokens and tokens[0] == '\*':

tokens.pop(0)

term = parse\_term(tokens)

return factor \* term

return factor

def parse\_factor(tokens):

if tokens and tokens[0] == '(':

tokens.pop(0)

expression = parse\_expression(tokens)

if not tokens or tokens.pop(0) != ')':

raise ParseError("Expected closing parenthesis")

return expression

if tokens and tokens[0].isnumeric():

return int(tokens.pop(0))

raise ParseError("Invalid expression")

def parse(input\_string):

tokens = list(input\_string.replace(" ", ""))

result = parse\_expression(tokens)

if tokens:

raise ParseError("Invalid expression")

return result

try:

input\_expression = "2 \* (3 + 4)"

result = parse(input\_expression)

print(f"Result: {result}")

except ParseError as e:

print(f"Parse Error: {e}")

**Output:**

Result: 14

**Exp-12: Implement an Earley parser for context-free grammars using a simple python program.**

**Code:**

import nltk

from nltk import CFG

from nltk.parse.earleychart import EarleyChartParser

grammar = CFG.fromstring("""

S -> NP VP

NP -> Det N

VP -> V NP | V

Det -> 'the' | 'a'

N -> 'cat' | 'dog'

V -> 'chased' | 'barked'

""")

parser = EarleyChartParser(grammar)

sentence = "the cat chased a dog"

tokens = nltk.word\_tokenize(sentence)

for tree in parser.parse(tokens):

tree.pretty\_print()

**Output:**

S

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

| VP

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

NP | NP

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

Det N V Det N

| | | | |

the cat chased a dog

**Exp-13:** **Generate a parse tree for a given sentence using a context-free grammar using python program.**

**Code:**

import nltk

from nltk import CFG

from nltk.parse import EarleyChartParser

grammar = CFG.fromstring("""

S -> NP VP

NP -> Det N | N

VP -> V NP

Det -> 'the' | 'a'

N -> 'cat' | 'dog' | 'ball'

V -> 'chased' | 'ate'

""")

sentence = "the cat chased the dog"

tokens = sentence.split()

parser = EarleyChartParser(grammar)

for tree in parser.parse(tokens):

tree.pretty\_print()

**Output:**

S

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

| VP

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

NP | NP

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

Det N V Det N

| | | | |

the cat chased the dog

**Exp-14:** **Create a program in python to check for agreement in sentences based on a context-free Grammar’s rules.**

**Code:**

import nltk

from nltk import CFG, ChartParser

grammar = CFG.fromstring("""

S -> NP\_SG VP\_SG | NP\_PL VP\_PL

NP\_SG -> 'cat' | 'dog'

NP\_PL -> 'cats' | 'dogs'

VP\_SG -> 'chases' | 'eats'

VP\_PL -> 'chase' | 'eat'

""")

parser = ChartParser(grammar)

def check\_agreement(sentence):

tokens = nltk.word\_tokenize(sentence)

tokens = [token for token in tokens if token in grammar.\_lexical\_index]

for tree in parser.parse(tokens):

return True

return False

sentences = [

"the cat chases",

"the cats chase",

"dogs eat",

"dog eats"

]

for sentence in sentences:

if check\_agreement(sentence):

print(f"'{sentence}' follows the grammar rules.")

else:

print(f"'{sentence}' violates the grammar rules.")

**Output:**

'the cat chases' follows the grammar rules.

'the cats chase' follows the grammar rules.

'dogs eat' follows the grammar rules.

'dog eats' follows the grammar rules.

**Exp-15:** **Implement probabilistic context-free grammar parsing for a sentence using python.**

**Code:**

import spacy

nlp = spacy.load("en\_core\_web\_sm")

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

doc = nlp(sentence)

for token in doc:

print(f"{token.text}: {token.dep\_} --> {token.head.text}")

**Output:**

The: det --> fox

quick: amod --> fox

brown: amod --> fox

fox: nsubj --> jumps

jumps: ROOT --> jumps

over: prep --> jumps

the: det --> dog

lazy: amod --> dog

dog: pobj --> over

.: punct --> jumps

**Exp-16:** **Implement a Python program using the SpaCy library to perform Named Entity Recognition (NER) on a given text.**

**Code:**

import spacy

nlp = spacy.load("en\_core\_web\_sm")

text = "Hi I am Jeff Bejos the owner of Amazon pvt lmtd"

doc = nlp(text)

for ent in doc.ents:

print(f"Entity: {ent.text}, Label: {ent.label\_}")

**Output:**

Entity: Jeff Bejos, Label: PERSON

Entity: Amazon, Label: ORG

**Exp-17:** **Write program demonstrates how to access WordNet, a lexical database, to retrieve synsets and explore word meanings in python.**

**Code:**

from nltk.corpus import wordnet

word = "car"

synsets = wordnet.synsets(word)

for synset in synsets:

print(f"Synset: {synset.name()}")

print(f"Definition: {synset.definition()}")

print(f"Examples: {', '.join(synset.examples())}")

print()

if synsets:

hypernyms = synsets[0].hypernyms()

if hypernyms:

print(f"Hypernyms for {synsets[0].name()}: {', '.join([h.name() for h in hypernyms])}")

if synsets:

hyponyms = synsets[0].hyponyms()

if hyponyms:

print(f"Hyponyms for {synsets[0].name()}: {', '.join([h.name() for h in hyponyms])}")

**Output:**

Synset: car.n.01

Definition: a motor vehicle with four wheels; usually propelled by an internal combustion engine

Examples: he needs a car to get to work

Synset: car.n.02

Definition: a wheeled vehicle adapted to the rails of railroad

Examples: three cars had jumped the rails

Synset: car.n.03

Definition: the compartment that is suspended from an airship and that carries personnel and the cargo and the power plant

Examples:

Synset: car.n.04

Definition: where passengers ride up and down

Examples: the car was on the top floor

Synset: cable\_car.n.01

Definition: a conveyance for passengers or freight on a cable railway

Examples: they took a cable car to the top of the mountain

Hypernyms for car.n.01: motor\_vehicle.n.01

Hyponyms for car.n.01: ambulance.n.01, beach\_wagon.n.01, bus.n.04, cab.n.03, compact.n.03, convertible.n.01, coupe.n.01, cruiser.n.01, electric.n.01, gas\_guzzler.n.01, hardtop.n.01, hatchback.n.01, horseless\_carriage.n.01, hot\_rod.n.01, jeep.n.01, limousine.n.01, loaner.n.02, minicar.n.01, minivan.n.01, model\_t.n.01, pace\_car.n.01, racer.n.02, roadster.n.01, sedan.n.01, sport\_utility.n.01, sports\_car.n.01, stanley\_steamer.n.01, stock\_car.n.01, subcompact.n.01, touring\_car.n.01, used-car.n.01

**Exp-18:** **Implement a simple FOPC parser for basic logical expressions using python program.**

**Code:**

from pyparsing import Word, alphas, infixNotation, opAssoc, Group, Suppress

variable = Word(alphas, exact=1)

constant = Word(alphas.upper(), exact=1)

and\_op = Suppress('&')

or\_op = Suppress('|')

not\_op = Suppress('~')

expr = infixNotation(

variable | constant,

[

(not\_op, 1, opAssoc.RIGHT),

(and\_op, 2, opAssoc.LEFT),

(or\_op, 2, opAssoc.LEFT),

],

)

def parse\_fopc(input\_string):

return expr.parseString(input\_string, parseAll=True)[0]

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

input\_string = "P & (Q | ~R)"

parsed\_expr = parse\_fopc(input\_string)

print(f"Input: {input\_string}")

print(f"Parsed: {parsed\_expr}")

**Output:**

Input: P & (Q | ~R)

Parsed: ['P', ['Q', ['R']]]

**Exp-19:** **Create a program for word sense disambiguation using the Lesk algorithm using python.**

**Code:**

from nltk.corpus import wordnet

from nltk.tokenize import word\_tokenize

from nltk.corpus import stopwords

def lesk\_algorithm(word, sentence):

best\_sense = None

max\_overlap = 0

word = word.lower()

context = set(word\_tokenize(sentence))

context = [w for w in context if w not in stopwords.words('english')]

for sense in wordnet.synsets(word):

definition = set(word\_tokenize(sense.definition())

+ [w for lemma in sense.lemmas() for w in lemma.name().split('\_')])

overlap = len(definition.intersection(context))

for example in sense.examples():

example\_words = set(word\_tokenize(example))

overlap += len(definition.intersection(example\_words))

if overlap > max\_overlap:

max\_overlap = overlap

best\_sense = sense

return best\_sense

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

word = "bank"

sentence = "I deposited my money in the bank by the river."

sense = lesk\_algorithm(word, sentence)

if sense:

print(f"Word: {word}")

print(f"Sense: {sense.name()}")

print(f"Definition: {sense.definition()}")

else:

print("No sense found.")

**Output:**

Word: bank

Sense: depository\_financial\_institution.n.01

Definition: a financial institution that accepts deposits and channels the money into lending activities

**Exp-20:** **Implement a basic information retrieval system using TF-IDF (Term Frequency-Inverse Document Frequency) for document ranking using python.**

**Code:**

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.metrics.pairwise import linear\_kernel

documents = [

"The quick brown fox jumps over the lazy dog",

"A brown dog jumped over a fox",

"The dog is quick and brown",

"The cat is black",

"A black cat and a brown dog are friends",

]

query = "quick brown fox"

tfidf\_vectorizer = TfidfVectorizer()

tfidf\_matrix = tfidf\_vectorizer.fit\_transform(documents)

query\_tfidf = tfidf\_vectorizer.transform([query])

cosine\_similarities = linear\_kernel(query\_tfidf, tfidf\_matrix)

document\_scores = list(enumerate(cosine\_similarities[0]))

document\_scores.sort(key=lambda x: x[1], reverse=True)

top\_n = 3

top\_documents = document\_scores[:top\_n]

for i, score in top\_documents:

print(f"Document {i + 1}: {documents[i]} (Score: {score:.2f})")

**Output:**

Document 1: The quick brown fox jumps over the lazy dog (Score: 0.50)

Document 2: A brown dog jumped over a fox (Score: 0.44)

Document 3: The dog is quick and brown (Score: 0.44)

**Exp-21:** **Create a python program that performs syntax-driven semantic analysis by extracting noun phrases and their meanings from a sentence.**

**Code:**

import nltk

from nltk.tokenize import word\_tokenize

from nltk.tag import pos\_tag

from nltk.corpus import wordnet

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

tokens = word\_tokenize(sentence)

pos\_tags = pos\_tag(tokens)

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

cp = nltk.RegexpParser(grammar)

tree = cp.parse(pos\_tags)

noun\_phrases = []

for subtree in tree.subtrees(filter=lambda t: t.label() == 'NP'):

noun\_phrases.append(' '.join(word for word, tag in subtree.leaves()))

for noun\_phrase in noun\_phrases:

words = word\_tokenize(noun\_phrase)

head\_noun = words[-1]

meanings = []

for synset in wordnet.synsets(head\_noun):

meanings.append(synset.definition())

print(f"Noun Phrase: '{noun\_phrase}'")

print(f"Meanings: {', '.join(meanings)}\n")

**Output:**

Noun Phrase: 'The quick brown'

Meanings: an orange of low brightness and saturation, Scottish botanist who first observed the movement of small particles in fluids now known a Brownian motion (1773-1858), abolitionist who was hanged after leading an unsuccessful raid at Harper's Ferry, Virginia (1800-1859), a university in Rhode Island, fry in a pan until it changes color, make brown in color, of a color similar to that of wood or earth, (of skin) deeply suntanned

Noun Phrase: 'fox'

Meanings: alert carnivorous mammal with pointed muzzle and ears and a bushy tail; most are predators that do not hunt in packs, a shifty deceptive person, the grey or reddish-brown fur of a fox, English statesman who supported American independence and the French Revolution (1749-1806), English religious leader who founded the Society of Friends (1624-1691), a member of an Algonquian people formerly living west of Lake Michigan along the Fox River, the Algonquian language of the Fox, deceive somebody, be confusing or perplexing to; cause to be unable to think clearly, become discolored with, or as if with, mildew spots

Noun Phrase: 'the lazy dog'

Meanings: a member of the genus Canis (probably descended from the common wolf) that has been domesticated by man since prehistoric times; occurs in many breeds, a dull unattractive unpleasant girl or woman, informal term for a man, someone who is morally reprehensible, a smooth-textured sausage of minced beef or pork usually smoked; often served on a bread roll, a hinged catch that fits into a notch of a ratchet to move a wheel forward or prevent it from moving backward, metal supports for logs in a fireplace, go after with the intent to catch

**Exp-22:** **Create a python program that performs reference resolution within a text.**

**Code:**

import spacy

nlp = spacy.load("en\_core\_web\_sm")

def resolve\_references(text):

doc = nlp(text)

resolved\_text = []

previous\_noun = None

for token in doc:

if token.pos\_ in ["NOUN", "PROPN"]:

previous\_noun = token.text

resolved\_text.append(token.text)

elif token.pos\_ == "PRON" and previous\_noun:

resolved\_text.append(previous\_noun)

else:

resolved\_text.append(token.text)

return ' '.join(resolved\_text)

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

text = "John is a software engineer. He loves coding. Mary is a data scientist. She is also passionate about her work."

resolved\_text = resolve\_references(text)

print(resolved\_text)

**Output:**

John is a software engineer . engineer loves coding . Mary is a data scientist . scientist is also passionate about scientist work .

**Exp-23:** **Develop a python program that evaluates the coherence of a given text.**

**Code:**

import nltk

from nltk.corpus import stopwords

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

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.metrics.pairwise import cosine\_similarity

text = """

Coherence in writing means that the ideas in each sentence and paragraph are connected and logical.

It helps the reader to follow your arguments and understand your point. Without coherence, the text may be confusing.

There are several ways to achieve coherence in your writing, such as using transition words and repeating key concepts.

"""

sentences = sent\_tokenize(text)

stop\_words = set(stopwords.words("english"))

word\_tokens = [word\_tokenize(sentence) for sentence in sentences]

filtered\_tokens = [[word for word in words if word.lower() not in stop\_words] for words in word\_tokens]

preprocessed\_sentences = [" ".join(words).lower() for words in filtered\_tokens]

vectorizer = TfidfVectorizer()

tfidf\_matrix = vectorizer.fit\_transform(preprocessed\_sentences)

cosine\_matrix = cosine\_similarity(tfidf\_matrix, tfidf\_matrix)

coherence\_score = cosine\_matrix.sum(axis=1).mean()

print("Coherence Score:", coherence\_score)

**Output:**

Coherence Score: 1.1285168907590968

**Exp-24:** **Create a python program that recognizes dialog acts in a given dialog or conversation.**

**Code:**

import nltk

conversation = [

"Hello, how are you?",

"I'm doing well, thank you.",

"Can you help me with my homework?",

"Sure, I'd be happy to help.",

"Great! I have some math problems to solve."

]

def recognize\_dialog\_acts(sentences):

dialog\_acts = []

for sentence in sentences:

tokens = nltk.word\_tokenize(sentence)

tagged = nltk.pos\_tag(tokens)

parsed = nltk.ne\_chunk(tagged)

dialog\_act = None

verbs = [word for word, pos in tagged if pos.startswith('V')]

modal\_verbs = [word for word, pos in tagged if pos == 'MD']

keywords = ["help", "homework", "problem", "solve"]

if "can" in modal\_verbs and any(keyword in sentence for keyword in keywords):

dialog\_act = "Request for Help"

elif any(verb in verbs for verb in ["help", "assist", "support"]) and "you" in tokens:

dialog\_act = "Offer Help"

elif any(verb in verbs for verb in ["solve", "answer"]) and any(keyword in sentence for keyword in keywords):

dialog\_act = "Task Acknowledgment"

else:

dialog\_act = "Other"

dialog\_acts.append((sentence, dialog\_act))

return dialog\_acts

dialog\_acts = recognize\_dialog\_acts(conversation)

for sentence, act in dialog\_acts:

print(f"Dialog Act: {act}\nSentence: {sentence}\n")

**Output:**

Dialog Act: Other

Sentence: Hello, how are you?

Dialog Act: Other

Sentence: I'm doing well, thank you.

Dialog Act: Offer Help

Sentence: Can you help me with my homework?

Dialog Act: Other

Sentence: Sure, I'd be happy to help.

Dialog Act: Task Acknowledgment

Sentence: Great! I have some math problems to solve.

**Exp-25:** **Utilize the GPT-3 model to generate text based on a given prompt. Make sure to install the OpenAI GPT-3 library in python implementation.**

**Code:**

import openai

api\_key = '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'

prompt = "Translate the following English text to French: 'Hello, how are you?'"

response = openai.Completion.create(

engine="text-davinci-002",

prompt=prompt,

max\_tokens=50

)

generated\_text = response.choices[0].text

print("Generated Text:")

print(generated\_text)

**Output:**

Generated Text:

French: "Bonjour, comment ça va ?"

**Exp-26:** **Implement a machine translation program using the Hugging Face Transformers library, translate English text to French using python.**

**Code:**

from transformers import MarianTokenizer, MarianMTModel

model\_name = "Helsinki-NLP/opus-mt-en-fr"

tokenizer = MarianTokenizer.from\_pretrained(model\_name)

model = MarianMTModel.from\_pretrained(model\_name)

english\_text = "Hello, how are you?"

input\_ids = tokenizer.encode(english\_text, return\_tensors="pt")

translated\_ids = model.generate(input\_ids, max\_length=50, num\_beams=5, early\_stopping=True)

translated\_text = tokenizer.decode(translated\_ids[0], skip\_special\_tokens=True)

print("English Text:", english\_text)

print("French Translation:", translated\_text)

**Output:**

English Text: Hello, how are you?

French Translation: Bonjour, comment ça va ?

**Model1 :: Exp-1**

**Model 2:**

**Implement a text classification program using the Naive Bayes algorithm to classify text documents into categories (e.g., spam detection)**

**Code:**

import nltk

from nltk.corpus import stopwords

from nltk.tokenize import word\_tokenize

from nltk.stem import PorterStemmer

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.model\_selection import train\_test\_split

from sklearn.naive\_bayes import MultinomialNB

from sklearn.metrics import accuracy\_score, confusion\_matrix, classification\_report

corpus = [

("Buy now and get a special discount!", "spam"),

("Meet you at the park later?", "not spam"),

("Congratulations, you've won a prize!", "spam"),

("Can we reschedule our meeting for tomorrow?", "not spam")

]

stop\_words = set(stopwords.words('english'))

stemmer = PorterStemmer()

def preprocess\_text(text):

text = text.lower()

words = word\_tokenize(text)

words = [word for word in words if word.isalpha()]

words = [word for word in words if word not in stop\_words]

words = [stemmer.stem(word) for word in words]

return ' '.join(words)

X, y = zip(\*corpus)

X = [preprocess\_text(text) for text in X]

tfidf\_vectorizer = TfidfVectorizer(max\_features=5000)

X\_tfidf = tfidf\_vectorizer.fit\_transform(X)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_tfidf, y, test\_size=0.2, random\_state=42)

classifier = MultinomialNB()

classifier.fit(X\_train, y\_train)

y\_pred = classifier.predict(X\_test)

accuracy = accuracy\_score(y\_test, y\_pred)

confusion = confusion\_matrix(y\_test, y\_pred)

classification\_rep = classification\_report(y\_test, y\_pred)

print(f"Accuracy: {accuracy \* 100:.2f}%")

print("Confusion Matrix:\n", confusion)

print("Classification Report:\n", classification\_rep)

**Output:**

Accuracy: 0.00%

Confusion Matrix:

[[0 1]

[0 0]]

Classification Report:

precision recall f1-score support

not spam 0.00 0.00 0.00 1.0

spam 0.00 0.00 0.00 0.0

accuracy 0.00 1.0

macro avg 0.00 0.00 0.00 1.0

weighted avg 0.00 0.00 0.00 1.0

**Model3:**

**Create a Python program to identify and extract key phrases or keywords from a given text using techniques such as TF-IDF.**

**The sentences = [ "Artificial intelligence (AI) is a field of computer science.","Machine learning is a subset of AI that focuses on training models to make predictions.", "Deep learning is a type of machine learning that uses neural networks with multiple layers.","Neural networks are composed of interconnected nodes called neurons.", "Recurrent neural networks (RNNs) are commonly used in natural language processing tasks.",]**

**Code:**

from sklearn.feature\_extraction.text import TfidfVectorizer

from nltk.corpus import stopwords

import nltk

sentences = [

"Artificial intelligence (AI) is a field of computer science.",

"Machine learning is a subset of AI that focuses on training models to make predictions.",

"Deep learning is a type of machine learning that uses neural networks with multiple layers.",

"Neural networks are composed of interconnected nodes called neurons.",

"Recurrent neural networks (RNNs) are commonly used in natural language processing tasks.",

]

stop\_words = set(stopwords.words('english'))

def preprocess(text):

words = nltk.word\_tokenize(text)

words = [word.lower() for word in words if word.isalnum() and word.lower() not in stop\_words]

return ' '.join(words)

preprocessed\_sentences = [preprocess(sentence) for sentence in sentences]

tfidf\_vectorizer = TfidfVectorizer()

tfidf\_matrix = tfidf\_vectorizer.fit\_transform(preprocessed\_sentences)

feature\_names = tfidf\_vectorizer.get\_feature\_names\_out()

tfidf\_scores = tfidf\_matrix.toarray()

for i, sentence in enumerate(sentences):

print(f"Keywords for Sentence {i + 1}: {sentence}")

keywords = [(feature\_names[j], tfidf\_scores[i][j]) for j in tfidf\_scores[i].argsort()[-5:][::-1]]

for keyword, score in keywords:

print(f"{keyword}: {score:.3f}")

print()

**Output:**

Keywords for Sentence 1: Artificial intelligence (AI) is a field of computer science.

intelligence: 0.421

artificial: 0.421

science: 0.421

computer: 0.421

field: 0.421

Keywords for Sentence 2: Machine learning is a subset of AI that focuses on training models to make predictions.

models: 0.355

predictions: 0.355

focuses: 0.355

training: 0.355

subset: 0.355

Keywords for Sentence 3: Deep learning is a type of machine learning that uses neural networks with multiple layers.

learning: 0.533

deep: 0.331

layers: 0.331

multiple: 0.331

uses: 0.331

Keywords for Sentence 4: Neural networks are composed of interconnected nodes called neurons.

nodes: 0.412

called: 0.412

composed: 0.412

interconnected: 0.412

neurons: 0.412

Keywords for Sentence 5: Recurrent neural networks (RNNs) are commonly used in natural language processing tasks.

language: 0.335

tasks: 0.335

commonly: 0.335

used: 0.335

rnns: 0.335

**Model4:**

**Write a Python program that utilizes NLTK's named entity recognition to extract named entities (e.g., person names, locations) from a given text. Write a Python program that performs reference resolution within a given text "Harvard University, located in Cambridge, Massachusetts, is a prestigious institution."**

**Code:**

import nltk

from nltk import word\_tokenize, pos\_tag, ne\_chunk

text = "Harvard University, located in Cambridge, Massachusetts, is a prestigious institution."

words = word\_tokenize(text)

tags = pos\_tag(words)

named\_entities = ne\_chunk(tags)

named\_entities\_list = []

for entity in named\_entities:

if isinstance(entity, nltk.Tree):

entity\_type = entity.label()

entity\_text = " ".join([word for word, tag in entity.leaves()])

named\_entities\_list.append((entity\_text, entity\_type))

print("Named Entities:")

for entity\_text, entity\_type in named\_entities\_list:

print(f"{entity\_type}: {entity\_text}")

resolved\_text = text.replace("Harvard University", "it").replace("located in Cambridge, Massachusetts", "there")

print("\nReference-Resolved Text:")

print(resolved\_text)

**Output:**

Named Entities:

ORGANIZATION: Harvard

GPE: University

GPE: Cambridge

GPE: Massachusetts

Reference-Resolved Text:

it, there, is a prestigious institution.