**Implement a python program that performs rule based part of speech tagging using regular expressions using the following rules**

**patterns = [**

**(r'\b(?:The|the)\b', 'DET'),**

**(r'\b(?:cat|dog)\b', 'NOUN'),**

**(r'\b(?:is|am|are)\b', 'VERB'),**

**(r'\b(?:quickly|brightly)\b', 'ADV'),**

**(r'\b(?:[A-Za-z]+)\b', 'NOUN')**

**]**

**CODE:**

import nltk

from nltk.tag import RegexpTagger

# patterns = [

# (r'^\d+$', 'CD'), # cardinal numbers

# (r'.\*ing$', 'VBG'), # gerunds

# (r'.\*ed$', 'VBD'), # past tense verbs

# (r'.\*es$', 'VBZ'), # 3rd person singular present tense verbs

# (r'.\*ould$', 'MD'), # modals

# (r'.\*\'s$', 'NN$'), # possessive nouns

# (r'.\*s$', 'NNS'), # plural nouns

# (r'^[A-Z][a-z]\*$', 'NNP'), # proper nouns

# (r'.\*', 'NN') # default noun

# ]

patterns = [

(r'\b(?:The|the)\b', 'DET'),

(r'\b(?:cat|dog)\b', 'NOUN'),

(r'\b(?:is|am|are)\b', 'VERB'),

(r'\b(?:quickly|brightly)\b', 'ADV'),

(r'.\*ed$', 'VERB'),

(r'.\*es$', 'VERB'),

(r'\b(?:[A-Za-z]+)\b', 'NOUN')

]

tagger = RegexpTagger(patterns)

words = ["the", "dog", "chased", "the", "cat"]

tagged\_words = tagger.tag(words)

print(tagged\_words)

**create a python program for sentiment analysis which can be positive,negative,or neutral on a set of text data"I love this product!It's amazing"**

**Code:**

import nltk

from nltk.sentiment.vader import SentimentIntensityAnalyzer

nltk.download('vader\_lexicon')

def analyze\_sentiment(text):

sid = SentimentIntensityAnalyzer()

sentiment\_scores = sid.polarity\_scores(text)

if sentiment\_scores['compound'] >= 0.05:

return 'Positive'

elif sentiment\_scores['compound'] <= -0.05:

return 'Negative'

else:

return 'Neutral'

text\_data = "I love this product! It's amazing."

sentiment\_result = analyze\_sentiment(text\_data)

print(f"Sentiment: {sentiment\_result}")

**Develop a python program that perform Named Entity Recoginition(NER) on the given text using popular libraries or models for the sentence "The capital of France is Paris, and it's known for the Eiffel Tower"**

**CODE:**

import spacy

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

def perform\_ner(text):

# Process the text using SpaCy

doc = nlp(text)

entities = [(ent.text, ent.label\_) for ent in doc.ents]

return entities

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

example\_text = "The capital of France is Paris, and it's known for the Eiffel Tower"

named\_entities = perform\_ner(example\_text)

print("Named Entities:")

for entity, label in named\_entities:

print(f"{entity} - {label}")

**Develop a python program for dependency parsing of a sentence using a dependency parser**

**CODE:**

import spacy

def dependency\_parsing(sentence):

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

doc = nlp(sentence)

for token in doc:

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

sentence1 = "John and Mary went to the store."

sentence2 = "The big brown dog chased the small black cat."

dependency\_parsing(sentence1)

print("\n" + "="\*30 + "\n") # Just to separate the outputs

dependency\_parsing(sentence2)

**write a python programs that employs the porter stemmer algorithm form python library to perform word stemming on a list of words. The sample sentences are "Coding with Python is very enjoyable.","I had a delicious meal at the restaurant"**

**code:**

import nltk

from nltk.stem import PorterStemmer

from nltk.tokenize import word\_tokenize

nltk.download('punkt')

def perform\_stemming(sentence):

porter = PorterStemmer()

words = word\_tokenize(sentence)

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

return stemmed\_words

sentences = [

"Coding with Python is very enjoyable.",

"I had a delicious meal at the restaurant"

]

for sentence in sentences:

stemmed\_words = perform\_stemming(sentence)

print(f"Original sentence: {sentence}")

print(f"Stemmed words: {' '.join(stemmed\_words)}\n")

**Write a python program that uses nltk to perfomr part of speech tagging on a given text "The sun is shining brightly","I love reading intersting books"**

**CODE:**

import nltk

from nltk.tokenize import word\_tokenize\

nltk.download('punkt')

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

def pos\_tagging(text):\

words = word\_tokenize(text)\

tagged\_words = nltk.pos\_tag(words)

return tagged\_words

texts = [

"The sun is shining brightly.",

"I love reading interesting books."

]

for text in texts:

tagged\_words = pos\_tagging(text)

print(f"Original text: {text}")

print(f"Part-of-speech tagged words: {tagged\_words}\n")

**implement a python program that perform Named Entity Recoginition(NER) on the given text using popular libraries or models for the sentence "Apple Inc. is headquartered in cupertino,California, and its CEO, Tim Cook, often delivers keynote speeches. The Eiffel Tower in Paris, France, is a popular tourist attraction."**

**CODE**:

import spacy

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

def perform\_ner(text):

doc = nlp(text)

entities = [(ent.text, ent.label\_) for ent in doc.ents]

return entities

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

example\_text = "Apple Inc. is headquartered in cupertino,California, and its CEO, Tim Cook, often delivers keynote speeches.The Eiffel Tower in Paris, France, is a popular tourist attraction."

# example\_text = "The capital of France is Paris, and it's known for the Eiffel Tower"

named\_entities = perform\_ner(example\_text)

print("Named Entities:")

for entity, label in named\_entities:

print(f"{entity} - {label}")

**Implement a simple first order predicate calculus FOPC parser for basic logical expresions using python. The varibles = {'p':True, 'q':True, 'r':False} and expressions = ['p and q','p or q','not p','q and (r or P)']**

**CODE:**

class Proposition:

def \_init\_(self, name, value):

self.name = name

self.value = value

class UnaryOperation:

def \_init\_(self, expression):

self.expression = expression

class BinaryOperation:

def \_init\_(self, left, right):

self.left = left

self.right = right

def parse\_expression(expression\_str):

expression\_str = expression\_str.replace("(", "( ").replace(")", " )")

tokens = expression\_str.split()

def parse(tokens):

token = tokens.pop(0)

if token == "not":

return UnaryOperation(parse(tokens))

elif token == "(":

left = parse(tokens)

operator = tokens.pop(0) # Get the operator ('and' or 'or')

right = parse(tokens)

tokens.pop(0) # Pop the closing parenthesis

return BinaryOperation(left, right) if operator == "and" else BinaryOperation(left, right)

else:

return Proposition(token, variables[token])

return parse(tokens)

def evaluate\_expression(expression, variables):

if isinstance(expression, Proposition):

return expression.value

if isinstance(expression, UnaryOperation):

return not evaluate\_expression(expression.expression, variables)

if isinstance(expression, BinaryOperation):

left = evaluate\_expression(expression.left, variables)

right = evaluate\_expression(expression.right, variables)

return left and right if isinstance(expression, BinaryOperation) else left or right

# Define the variables and their values

variables = {

"p": True,

"q": True,

"r": False

}

expressions = [

"p and q",

"p or q",

"not p",

"q and ( r or p )"

]

for expression\_str in expressions:

expression = parse\_expression(expression\_str)

result = evaluate\_expression(expression, variables)

print(f"Expression: {expression\_str}, Result: {result}")

**Implement a machine translation programs using the hugging face transformers library in Python using model "Helsinki-NLP/opus-mt-en-de" to translate English to german**

**code:**

from transformers import AutoTokenizer, AutoModelForSeq2SeqLM

import torch

tokenizer = AutoTokenizer.from\_pretrained("Helsinki-NLP/opus-mt-en-de")

model = AutoModelForSeq2SeqLM.from\_pretrained(

"Helsinki-NLP/opus-mt-en-de").to("cuda" if torch.cuda.is\_available() else "cpu")

def translate(text):

encoded\_text = tokenizer(text, return\_tensors="pt")

output = model.generate(\*\*encoded\_text)

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

return decoded\_text

sentence = "Hello, how are you?"

translated\_sentence = translate(sentence)

print(translated\_sentence)

**Develop a program for semantic analysis(positive,negative or neutral)using textblob library(eg., using a pre-trained model or building a custom model),to the sentences "I love this product! It's amazing.","The weather is terrible today."**

**Code:**

from textblob import TextBlob

def perform\_sentiment\_analysis(sentence):

blob = TextBlob(sentence)

sentiment\_polarity = blob.sentiment.polarity

if sentiment\_polarity > 0:

return 'Positive'

elif sentiment\_polarity < 0:

return 'Negative'

else:

return 'Neutral'

sentence1 = "I love this product! It's amazing."

sentence2 = "The weather is terrible today."

result1 = perform\_sentiment\_analysis(sentence1)

result2 = perform\_sentiment\_analysis(sentence2)

print(f"Sentence 1: {sentence1}")

print(f"Sentiment 1: {result1}")

print()

print(f"Sentence 2: {sentence2}")

print(f"Sentiment 2: {result2}")

**Implement a Python-based named entity disambiguation program that resolves entity mentions to their corresponding Wikipedia entities using wikipediaapi. Use the following input sentences "Apple is a leading tech company.""I love apples as a fruit.",**

**"Python is a popular programming language.". "The python is a non-venomous snake**

**CODE :**

import wikipediaapi

import re

import requests

def disambiguate\_entities(sentence):

user\_agent = "Your-User-Agent-Name/1.0 (your@email.com)"

entity\_mentions = re.findall(r'\b\w+\b', sentence)

resolved\_entities = {}

for mention in entity\_mentions:

headers = {'User-Agent': user\_agent}

url = f'https://en.wikipedia.org/wiki/{mention}'

response = requests.get(url, headers=headers)

if response.status\_code == 200:

resolved\_entities[mention] = response.url.split('/')[-1]

else:

resolved\_entities[mention] = None

return resolved\_entities

sentence1 = "Apple is a leading tech company."

sentence2 = "I love apples is a fruit."

sentence3 = "Python is a popular programming language."

sentence4 = "The python is a non-venemous snake."

result1 = disambiguate\_entities(sentence1)

print("Entities in sentence 1:", result1)

result2 = disambiguate\_entities(sentence2)

print("Entities in sentence 2:", result2)

result3 = disambiguate\_entities(sentence3)

print("Entities in sentence 3:", result3)

result4 = disambiguate\_entities(sentence4)

print("Entities in sentence 4:", result4)

**Implment a basic information retrieval system using TF-IDF for document ranking in Python using the sentences = ["Climate change is a pressing global issue that requires immediate action.",**

**"Renewable energy sources, such as solar and wind power, are essential for reducing carbon emissions.",**

**"Greenhouse gases, like carbon dioxide and methane, contribute to global warming.",**

**"The Paris Agreement is an international treaty aimed at addressing climate change.",**

**"Sustainablity and environmental conservation are crucial for the future of our planet."]**

**CODE:**

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.metrics.pairwise import cosine\_similarity

sentences = [

"Climate change is a pressing global issue that requires immediate action.",

"Renewable energy sources, such as solar and wind power, are essential for reducing carbon emissions.",

"Greenhouse gases, like carbon dioxide and methane, contribute to global warming.",

"The Paris Agreement is an international treaty aimed at addressing climate change.",

"Sustainability and environmental conservation are crucial for the future of our planet."

]

vectorizer = TfidfVectorizer()

tfidf\_matrix = vectorizer.fit\_transform(sentences)

def rank\_documents(query, sentences, tfidf\_matrix, vectorizer):

query\_vector = vectorizer.transform([query])

similarity\_scores = cosine\_similarity(query\_vector, tfidf\_matrix)

ranked\_sentences = sorted([(idx, score) for idx, score in enumerate(

similarity\_scores[0])], key=lambda x: -x[1])

return ranked\_sentences

query = "Climate change and renewable energy"

ranked\_docs = rank\_documents(query, sentences, tfidf\_matrix, vectorizer)

print("Ranked Documents:")

for idx, score in ranked\_docs:

print(f"Similarity Score: {score:.4f}, Sentence: {sentences[idx]}")

**Write a python program that takes a text and a regular expression pattern as input and demonstrates how to use regular expressions to find and extract specific patterns in the text. using pattern = r'\b\w{3}\b' and input sentence "The quick brown fox jumps over the lazy dog. The cat is also agile."**

**CODE**:

import re

def find\_patterns(text, pattern):

matches = re.findall(pattern, text)

return matches

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

input\_sentence = "The quick brown fox jumps over the lazy dog. The cat is also agile."

pattern = r'\b\w{3}\b' # Matches three-letter words

extracted\_patterns = find\_patterns(input\_sentence, pattern)

print(f"Input sentence: {input\_sentence}")

print(f"Pattern: {pattern}")

print(f"Extracted patterns: {extracted\_patterns}")

**create a python program to recogniz dialog acts in a given dialog**

**conversation = [**

**"Hello! How are you today?",**

**"I'm doing well, thank you. How about you?",**

**"Can you please pass the salt?",**

**"Sure, here you go.",**

**"What time is the meeting tomorrow?",**

**"The meeting is at 2:00 PM."**

**]**

**CODE :**

import spacy

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

dialog\_acts = []

conversation = [

"Hello! How are you today?",

"I'm doing well, thank you. How about you?",

"Can you please pass the salt?",

"Sure, here you go.",

"What time is the meeting tomorrow?",

"The meeting is at 2:00 PM."

]

for utterance in conversation:

doc = nlp(utterance)

if any(token.text.lower() in ['hello', 'hi', 'hey'] for token in doc):

dialog\_acts.append("Greeting")

elif any(token.pos\_ == 'AUX' or token.text.lower() in ['what', 'where', 'when', 'how', 'can', 'please', 'pass', 'time'] for token in doc):

dialog\_acts.append("Question")

else:

dialog\_acts.append("Statement")

print(dialog\_acts)

**Implement a python program that construct a finite state automation capable of recognizing dates in a specific format**

**CODE:**

states = ["start", "day", "slash1", "month", "slash2", "year", "accept"]

current\_state = "start"

date\_input = "25/11/2023"

for char in date\_input:

if current\_state == "start" and char.isdigit():

current\_state = "day"

elif current\_state == "day" and char.isdigit():

current\_state = "slash1"

elif current\_state == "slash1" and char == "/":

current\_state = "month"

elif current\_state == "month" and char.isdigit():

current\_state = "slash2"

elif current\_state == "slash2" and char == "/":

current\_state = "year"

elif current\_state == "year" and char.isdigit():

current\_state = "accept"

else:

current\_state = "start"

result = current\_state == "accept"

print(f"Is '{date\_input}' a valid date? {result}")

**Write a python program that uses the NLTK library to perform morphological analysis on sentence "unhappily,she can ran quickly"**

**Code:**

import nltk

from nltk import word\_tokenize

from nltk import pos\_tag

nltk.download('punkt')

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

sentence = "Unhappily, she ran quickly"

words = word\_tokenize(sentence)

pos\_tags = pos\_tag(words)

print("Original Sentence:", sentence)

print("POS Tags:", pos\_tags)

**Develop a program for semantic analysis(positive,negative or neutral)using textblob library(eg., using a pre-trained model or building a custom model),to the sentences "I love this product! It's amazing.","The weather is terrible today."**

**Code:**

import nltk

from textblob import TextBlob

nltk.download('punkt')

def analyze\_sentiment(text):

blob = TextBlob(text)

sentiment\_polarity = blob.sentiment.polarity

if sentiment\_polarity > 0:

return "Positive"

elif sentiment\_polarity < 0:

return "Negative"

else:

return "Neutral"

sentence1 = "I love this product! It's amazing."

sentence2 = "The weather is terrible today."

print("Sentence 1:", analyze\_sentiment(sentence1))

print("Sentence 2:", analyze\_sentiment(sentence2))